Pure Subtype Systems:
A Type Theory for Extensible Software

DeLesley Hutchins

Doctor of Philosophy

Laboratory for Foundations of Computer Science

School of Informatics

University of Edinburgh

2009

Abstract

This thesis presents a novel approach to type theory called "pure subtype systems", and a core calculus called DEEP which is based on that approach. DEEP is capable of modeling a number of interesting language techniques that have been proposed in the literature, including mixin modules, virtual classes, feature-oriented programming, and partial evaluation.

The design of DEEP was motivated by two well-known problems: "the expression problem", and "the tag elimination problem." The expression problem is concerned with the design of an interpreter that is extensible, and requires an advanced module system. The tag elimination problem is concerned with the design of an interpreter that is efficient, and requires an advanced partial evaluator. We present a solution in DEEP that solves both problems simultaneously, which has never been done before.

These two problems serve as an "acid test" for advanced type theories, because they make heavy demands on the static type system. Our solution in DEEP makes use of the following capabilities. (1) *Virtual types* are type definitions within a module that can be extended by clients of the module. (2) Type definitions may be mutually recursive. (3) Higher-order subtyping and bounded quantification are used to represent partial information about types. (4) Dependent types and singleton types provide increased type precision.

The combination of recursive types, virtual types, dependent types, higher-order subtyping, and bounded quantification is highly non-trivial. We introduce "pure subtype systems" as a way of managing this complexity. Pure subtype systems eliminate the distinction between types and objects; every term can behave as either a type or an object depending on context. A subtype relation is defined over all terms, and subtyping, rather than typing, forms the basis of the theory. We show that higher-order subtyping is strong enough to completely subsume the traditional type relation, and we provide practical algorithms for type checking and for finding minimal types.

The cost of using pure subtype systems lies in the complexity of the meta-theory. Unfortunately, we are unable to establish some basic meta-theoretic properties, such as type safety and transitivity elimination, although we have made some progress towards these goals. We formulate the subtype relation as an abstract reduction system, and we show that the type theory is sound if the reduction system is confluent. We can prove that reductions are locally confluent, but a proof of global confluence remains elusive.

In summary, pure subtype systems represent a new and interesting approach to type theory. This thesis describes the basic properties of pure subtype systems, and provides concrete examples of how they can be applied. The Deep calculus demonstrates that our approach has a number of real-world practical applications in areas that have proved to be quite difficult for traditional type theories to handle. However, the ultimate soundness of the technique remains an open question.

Acknowledgements

I would first of all like to thank my supervisor, Philip Wadler, for all the help that he has given me during my time here at the University of Edinburgh. Phil was not always convinced that my approach to type theory was the right one to take, and his initial skepticism has turned out to be well-founded in many respects. However, he was willing to give me the freedom to pursue my own path, and make my own discoveries and mistakes. Yet Phil did not just leave to my own devices; he was always there to discuss my ideas, read my drafts, send me pointers to seminars, talks, and related work, and offer suggestions for improvement. I am very grateful for his input.

I would also like to thank the two examiners for my thesis, David Aspinall and Joe Wells, for taking the time to go through this entire document with a fine-toothed comb. David and Joe have really helped me to improve the thesis, by pointing out everything from minor corrections to major clarifications. In addition, I would like to thank Dave Robertson, my second supervisor, for his help and support. Dave was always there to provide encouragement, advice, and a sympathetic ear.

My heartfelt thanks go out too to Bob Praus and Steve Coy of MZA Associates Corporation. I was employed by MZA before starting my PhD, and many of the ideas in this thesis were inspired by work that I initially did for the company. When I decided to pursue those ideas further in grad school, MZA generously offered to fund my research. Without their financial support, this work would never have been possible.

As with any scientist, I have been encouraged and supported by many people in the research community in which I work. Don Batory proofread drafts of several papers and offered many suggestions. William Cook championed my first major publication at OOPSLA. Sven Apel worked with me as a coauthor on applications of the DEEP calculus. Conor McBride sat down with me to explain the mysterious world of dependently-typed pattern matching. Benjamin Pierce was there to answer several questions on the more subtle aspects of subtyping. Vincent Van Oostrom helped me work through the confluence proofs at the core of System $\lambda_{\triangleleft}$. Randy Pollack helped me with my proof techniques, and mechanically verified some of my proofs. I'm sure that there are others that I have forgotten to mention; my thanks go out to them as well.

In addition, I would like to thank my mother for being there for me during my studies. She was the one who initially suggested that I go back to school, and then was shocked when I announced that I would be moving across the Atlantic to pursue a degree in Scotland. But she saved my mail for me, stored my possessions, and spoke to me across the miles with a friendly voice and encouraging words in phone and e-mail.

Last but not certainly not least, I offer grateful thanks to my wife, Laura, who has been my constant companion throughout my PhD. We met within days of my arrival in Edinburgh, fell in love, got engaged, and then got married, in what was almost a storybook romance. She has helped motivate me when I needed it, propped me up when I was down, and has shared both my triumphs and my disappointments. Her love and commitment has kept me going; I couldn't have done this without her.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*DeLesley Hutchins*)

Table of Contents

List of Figures

Chapter 1

Introduction

> *If it could be demonstrated that any complex organ existed, which could not possibly have been formed by numerous, successive, slight modifications, my theory would absolutely break down. But I can find out no such case.*
>
> — Charles Darwin, *On the Origin of Species*

1.1 A tale of two problems

A piece of software is not a static entity, it is a dynamic entity which changes over time. Almost every software project starts out with a simple working prototype, which is then progressively improved by "numerous, successive, slight modifications." Modifications are necessary to eliminate bugs, add new features, or alter program behavior to meet changing requirements and needs. Although it is possible to do major software rewrites, and it is occassionally necessary, such rewrites are costly. In general, a well-designed piece of software is structured in such a way that it anticipates future change, and can be easily modified and extended.

Building extensible software is still more of an art than a science, and it depends primarily on the experience and skill of software engineers. (There is old joke that a Fortran programmer can write Fortran code in any language.) Nevertheless, programming languages also have a large role to play: a language should provide abstractions that make the development of extensible software easier and more convenient.

This thesis explores two areas in which mainstream programming languages fail to provide abstractions that are powerful enough. The failure of mainstream languages is illustrated by two well-known problems: *the expression problem* and *the tag-elimination problem*.

The expression problem is concerned with code factorization. It illustrates a case in which the code cannot be factored effectively into separate modules, because the underlying language does provide the necessary abstractions to do so. The standard formulation of the problem describes the design of an extensible interpreter for a toy language. However, the fundamental problem has much wider applicability; the basic elements require only a set of mutually recursive data types, and a set of operations defined by recursion on those data types. This situation is so common as to be almost universal, and yet no mainstream language adequately addresses it. Current languages make it easy to either extend the data types, or the set of operations, but not to extend both at the same time.

The tag-elimination problem is concerned with the efficiency of domain-specific libraries. Like the expression problem, the basic formulation describes an interpreter for a toy language. Partial evaluation has the potential to eliminate the overhead of interpretation, thus transforming the interpreter into a compiler. Automatically generating compilers from interpreters is important because it allows a domain-specific language to be efficiently embedded in a general-purpose one, with broad application for optimized software libraries. Unfortunately, partial evaluation is not able to eliminate interpretive overhead in certain cases.

Both of these problems have been widely discussed in the literature, and many solutions have been proposed for each of them. However, the two problems have been studied by separate research communities, and as a result, there is little overlap between existing proposals. We argue that although the two problems look quite different on the surface, it is quite natural to combine them together. We refer to this combination as *the DSL expression problem*. The DSL expression problem is concerned with the design of an interpreter that is both extensible and efficient, and to our knowledge, it cannot be solved by any existing language.

This thesis presents a formal calculus called DEEP, and its associated type theory, and shows how a solution to the DSL expression problem can be written in DEEP. We call our calculus DEEP because it supports *deep mixin composition*, one of the critical techniques which allows the expression problem to be solved. We have a prototype implementation of the DEEP calculus, type system, and partial evaluator which demonstrates that our solution works.

Our ultimate goal in the design of DEEP was to develop a general-purpose language that is truly extensible, one that would allow us to seamlessly embed other domain-specific languages inside of it. In order for an embedding to be “seamless”, it must have

several characteristics. First, the embedding should be efficient: there should be no difference in execution speed between code written in the domain-specific language, and code written directly in DEEP. Second, the embedding should be type-safe: the type-checking phase for DEEP should catch type errors in the domain-specific language. Third, the domain-specific language itself should be extensible with new constructs. The "DSL Expression Problem" captures these requirements.

1.1.1 Pure subtype systems

The most difficult part of our solution, by far, is the type theory. Both the expression problem and the tag-elimination problem are relatively easy to solve in dynmically typed languages like Lisp and Scheme. The tag-elimination problem in particular doesn't even arise in a dynamically-typed language at all. However, both problems are difficult to solve in a statically typed language, because they require an advanced type system.

A solution to the expression problem requires first-class mixin modules with subtyping, bounded quantification on types, and late binding for recursive types. This combination of features is difficult work with, and is supported by only a few existing module calculi. The tag elimination problem additionally requires either dependent types or generalized abstract data types (GADTs), which interact with the module system in a non-trivial way.

We propose *pure subtype systems* as a general technique for managing this complexity. Pure subtype systems eliminate the distinction between types and objects; every term can behave as either a type or an object depending on context. A subtype relation is defined over all terms, and subtyping, rather than typing, forms the basis of the theory.

Pure subtype systems are simpler than traditional type systems because there is one relation: subtyping, rather than two relations: typing and subtyping. They are also expressive, because language of types is just as powerful as the language of objects. The immediate consequence of expressiveness is that type checking is undecidable. Moreover, the meta-theory is somewhat intractable as well; we have thus far been unable to prove basic properties like type safety and transitivity elimination, although we have made some progress towards these goals.

Despite the metatheoretic difficulties, we are able to show that the type theory is usable in practice by describing practical algorithms for type-checking. These algorithms

are used in the prototype implementation of DEEP.

1.2 The expression problem

The basic issues that underly the expression problem were identified by Reynolds more than 30 years ago, and have been discussed at length by many people since then [Reynolds, 1975] [Cook, 1990] [Krishnamurthi et al., 1998] [Zenger and Odersky, 2001] [Bruce, 2003] [Ernst, 2003] [Torgersen, 2004] [Lopez-Herrejon et al., 2005] [Zenger and Odersky, 2005]. The name "the expression problem", and the formulation of the problem that we use here is due to Wadler [Wadler, 1998].

The expression problem is concerned with the design of a simple interpreter or compiler. The job of the interpreter is to manipulate expressions in a language L of some kind (hence the name "expression problem"). Expressions are represented by an abstract syntax tree, or AST. The data type for the AST defines the syntax of the language L, and consists of a number of different variants for the various terminals and non-terminals that may appear in the AST (e.g. integer literals, "$+$", "$\times$", etc.). The intepreter itself is implemented as a set of operations that may traverse the AST. Possible operations include evaluation, typing, pretty-printing, optimization, and so on.

Given this structure, there are two natural ways in which one might wish to extend the interpreter. First, one might wish to extend the language L that it interprets; such an extension involves adding a new variant to the AST data type, and updating all of the operations to recognize the new variant. Second, one might wish to add a new operation over ASTs, such as adding a type-checking routine to an untyped interpreter. Both of these extensions are easy enough to do by modifying the source code of the interpreter. However, the expression problem specifically dictates that direct modification of source code is not allowed.

There are many reasons why direct modification of source code is not desirable. The interpreter might be distributed as a third-party library, in which case the code would be unavailable. Even if the source code were available, it might be written and maintained by a different group of engineers. Perhaps most importantly, different users of the interpreter might wish to extend it in different ways. In all of these cases, an extension must be considered separately from the original definition of the interpreter.

In an object-oriented language, it is easy to add new variants, but hard to add new operations. In a functional language, it is easy to add new operations, but hard to add

```
// basic interpreter definition
abstract class Expr {
  abstract int eval();
};

class Lit extends Expr {
  int litval;
  Lit(int v) { litval = v;    }
  int eval() { return litval; }
};

class Plus extends Expr {
  Expr left, right;
  Plus(Expr l, Expr r) { left = l; right = r; }
  int eval() { return left.eval() + right.eval(); }
};

// adding a new operation
interface ExprPrint {  // pretty printing
  String toString();
};

class LitPrint extends Lit implements ExprPrint {
  LitPrint(int v) { super(v); }
  String toString() { return String.valueOf(litval); }
};

class PlusPrint extends Plus implements ExprPrint {
  PlusPrint(ExprPrint l, ExprPrint r) { super(l,r); }
  String toString() {
    return "(" + ((ExprPrint) left).toString()  + "+" +
                 ((ExprPrint) right).toString() + ")";
  }
};
```

Figure 1.1: The expression problem in Java

new variants. We will illustrate this basic dilemma by showing a simple interpreter written in both Java and Haskell.

1.2.1 The expression problem: OO style

The interpreter design pattern [Gamma et al., 1995] describes how simple interpreters can be implemented in an object-oriented language. The first half of Figure 1.1 illustrates the pattern. Class Expr is the base class of expressions, and each variant is

```
data Expr = Lit Int
          | Plus Expr Expr

eval :: Expr -> Int
eval (Lit i)    = i
eval (Plus l r) = (eval l) + (eval r)

toString :: Expr -> String
toString (Lit i)    = show i
toString (Plus l r) = "(" ++ (toString l) ++ "+" ++ (toString r) ++ ")"
```

Figure 1.2: A simple interpreter in Haskell

---

implemented as a subclass of Expr. All operations must be declared as methods of Expr, and implemented within each subclass.

This pattern makes it easy to add a new variant, because a new variant is just a new class. (It is possible to add a new class without editing the source code of any existing classes.) However, it is hard to add a new operation, because all operations have to be declared within Expr, and the source code of Expr is not available.

The code in Figure 1.1 resolves the problem by creating a new interface (ExprPrint) which declares a new operation (toString). It then derives subclasses that implement ExprPrint. However, this solution is not statically safe because it involves dynamic type casts. Type casts are required because the type of left and right in PlusPrint is still Expr, not ExprPrint. Although the constructor of PlusPrint ensures that all subterms are printable, this is not reflected in the type of left and right.

One might attempt to make the interpreter type-safe by using F-bounded polymorphism to parameterize Expr with its "self-type", as described in [Torgersen, 2004]. As Torgersen notes, however, this approach is quite complex, and it does not scale to large class libraries with a number of mutually recursive classes, because each class would have to be parameterized by every class that it depends on.

1.2.2 The expression problem: functional programming style

Figure 1.2 shows how a simple interpreter could be implemented in Haskell. It is significantly simpler than the Java version, because functional languages were designed for recursive operations on variant data types. In Haskell, it is easy to add a new operation; the toString method is merely added as a second function. However, Haskell provides no way to add a new variant without modifying the definition of Expr and eval.

Swierstra has shown that it is possible to solve the expression problem in Haskell by abandoning Haskell's built-in data type and pattern matching mechanisms [Swierstra, 2008]. Swierstra defines his own set of type constructors which allow extension, and then uses Haskell's powerful type class mechanism to define operations on those types. Like Torgersen's solution in Java, Swierstra's solution requires a sophisticated coding protocol.

We argue that solutions which require a complex design pattern or coding protocol are flawed for two reasons. First, they require the programmer to specifically anticipate the need for extension before program development even begins. Second, a complex protocol is both difficult to design, and difficult to use. We feel that languages should provide mechanisms that allow programmers to structure code in whatever way seems most natural, without sacrificing extensibility.

1.2.3 The solution: late-binding for types

In most mainstream languages, including Java and Haskell, types use *early binding*. All classes in Java belong to a single global namespace. When a class name appears within a declaration, the compiler will resolve the name immediately, and replace it with a pointer to the global class definition.

Because names are bound early, relationships between classes are hard-coded. The Expr, Lit and Plus classes together comprise a *class family* [Ernst, 2001], in which the classes are related in a particular way: Lit and Plus derive from Expr, and Plus contains data members of type Expr. Since these relationship are hard-coded, it is not possible to extend the classes without breaking the relationships between them.

In object-oriented languages like Java, the use of early binding for class names contrasts sharply with the use of late binding for method names. The name of a method is not bound immediately at the point where it is declared. Instead, object-oriented languages use *late binding* for methods; the actual method that a name refers to is not resolved until run-time. Late binding is the basis for inheritance in OO languages; a derived class may override the definition of a method without breaking any code that calls the method.

It is quite easy to solve the expression problem by extending the mechanism of late binding so that it applies to classes as well as methods. Figure 1.3 shows what an object-oriented solution to the problem might look like in Java-like psuedocode. The Expr, Lit and Plus classes are declared together as a family, and then extended as a

```
// basic interpreter definition
module MExpr {
  abstract class Expr {
    abstract int eval();
  };

  class Lit extends Expr {
    int litval;
    Lit(int v) { litval = v;    }
    int eval() { return litval; }
  };

  class Plus extends Expr {
    Expr left, right;
    Plus(Expr l, Expr r) { left = l; right = r; }
    int eval() { return left.eval() + right.eval(); }
  };
}

// extension
module MPrint extends MExpr {
  class Expr extends super.Expr {
    String toString();
  };

  class Lit extends super.Lit {
    LitPrint(int v) { super(v); }
    String toString() { return String.valueOf(litval); }
  };

  class Plus extends super.Plus {
    PlusPrint(Expr l, Expr r) { super(l,r); }
    String toString() {
      return "(" + left.toString() + "+" + right.toString() + ")";
    }
  };
}
```

Figure 1.3: A solution to the expression problem in Java-like pseudocode

---

family, thus preserving the relationships between them. Inheritance between modules works exactly like inheritance between classes; the MPrint module inherits from MExpr, and overrides Expr, Lit and Plus with new definitions, where the new definitions inherit from the old ones.

It is important to note that MPrint.Lit is a subclass of MPrint.Expr. MPrint.Lit inher-

its from super.Lit, which refers to the definition of Lit within the MExpr module. The definition of Lit within MExpr inherits from this.Expr. Just like late binding for methods, the this keyword refers to "the current module", which is now MPrint. Because of late-binding, super.Lit thus inherits from MPrint.Expr, not MExpr.Expr.

In the object-oriented world, classes with late-binding are also known as *virtual classes*, by analogy with virtual methods. Virtual classes are an old idea, dating back to the Beta programming language [Madsen and Møller-Pedersen, 1989]. The above solution cannot be implemented in Beta, because Beta does not allow inheritance from a virtual class, but it can be implemented in gbeta, which is a generalization of Beta [Ernst, 1999b].

Virtual classes were largely ignored for almost a decade, but there has been a flurry of interest in recent years [Ernst, 2001] [Ernst, 2003] [Nystrom et al., 2005] [Jolly et al., 2005] [Igarashi et al., 2005] [Clarke et al., 2007]. Interest in virtual classes has coincided with the rise of aspect-oriented programming [Kiczales et al., 2001] and feature-oriented programming [Peri Tarr, 1999] [Batory et al., 2003], which are alternative mechanisms that address what is essentially the same problem.

Following Zenger and Odersky [Zenger and Odersky, 2005], we refer to this basic technique as *deep mixin composition*. The technique can be applied to any module system; it is not restricted to OO classes. When using deep mixin composition, refining a module (or a class) involves refining sub-modules (or nested classes), sub-sub-modules, and so on, down to any arbitrary depth. This ability to refine nested modules is what distinguishes deep mixin composition from ordinary OO inheritance, which is shallow. (Mainstream object-oriented languages like C++, Java, and C# do support nested classes, but they do not allow nested classes to be refined in this way.)

Virtual classes, features, aspects, and deep-mixin composition are all geared towards programming-in-the-large; they extend the mechanism of inheritance so that it can be used to manipulate whole class or module hierarchies rather than just individual classes.

1.2.4 The solution: functional programming style

Although virtual classes have been explored mostly in the context of object-oriented languages, the idea of late-binding for types works equally well in functional languages. Figure 1.4 shows a solution to the expression problem in Haskell-like pseudo-code.

```
// basic interpreter definition
module MExpr {
  data Expr = Lit Int
            | Plus Expr Expr

  eval :: Expr -> Int
  eval (Lit i)    = i
  eval (Plus l r) = (eval l) + (eval r)
}

module MPrint extends MExpr {
  data Expr = super.Expr
            | Times Expr Expr

  eval :: Expr -> Int
  eval (Times l r) = (eval l) * (eval r)
  eval e           = super.eval(e)
}
```

Figure 1.4: A solution to the expression problem in Haskell-like pseudocode

---

The Haskell solution, like the one before, encapsulates definitions within a module, and then uses module inheritance to extend all definitions within the module, including the definition of Expr. Adding a new variant to Expr also involves adding a new pattern to eval, so the new module overrides eval, and uses a default case to forward unhandled patterns to the previous definition. Variations on this solution can be found in [Zenger and Odersky, 2001] and [Duggan and Sourelis, 1996].

1.2.5 Type system requirements

It is possible to solve the expression problem by purely syntactic means. Aspect weaving [Kiczales et al., 2001], feature composition [Batory et al., 2003], and C++ templates [Smaragdakis and Batory, 2002] all provide code transformations that will merge the text of two modules together. The main challenge of the expression problem is the type system. We wish to design a static type system that is capable of type-checking and compiling each extension separately.

In order to solve the expression problem, the type system must support three things:

- **Virtual types** (i.e. late binding for types): Within a module, type members must be virtual, i.e. they must use late-binding. Virtual types are conceptually quite similar to ordinary polymorphism; the type name is treated as an unknown

variable when type-checking the module.

- **Recursion:** The Expr data type and the eval method are both recursive definitions. The module system must therefore support recursive modules. Recursive modules are much more complex than non-recursive modules, especially because recursion and late-binding interact in non-trivial ways.
- **Subtyping with bounded quantification:** The definition of a virtual type is not completely unknown. In the MExpr module, the precise definition of Expr is not known until run-time, but we do know that it has an eval method. In the Haskell version, we know that Expr has the Lit and Plus variants. There must be some way of representing partial information about unknown types. Subtyping with bounded quantification, as found in System $F_{\leq}$, is the technique that we choose in this thesis.
- **Dependent types:** Virtual types differ from ordinary polymorphism because they are members of a module, and can thus be extracted from the module using paths, such as MExpr.Lit. If modules are treated as first-class objects, then such paths require dependent types: MExpr.Lit is a type that depends on the object MExpr.

In addition to the requirements listed above, we follow [Zenger and Odersky, 2005] in demanding that the module system support multiple inheritance of some kind, so that it is possible to define two extensions independently, and then combine those two extensions together.

1.3 Doman specific languages and tag-elimination

Every problem has a particular problem domain: a set of concepts and techniques that are suitable for reasoning about that particular kind of problem. When solving a problem, a software engineer must translate domain-specific concepts into the general-purpose abstractions provided by the programming language. Abstractions such as procedures, functions, and classes are useful precisely because they are effective ways of representing concepts in a wide variety of problem domains.

There are times, however, when the concepts in a problem domain do not map precisely onto the abstractions provided by the programming language. For example, one way to represent a parser is as a grammar in Backus-Naur form, where production rules have been annotated with actions. However, grammars and production rules do not directly correspond to classes and methods. A parser generator like yacc [Levine

et al., 1992] remedies this mismatch by providing a domain-specific language (DSL) that translates grammars and rules into classes and methods.

1.3.1 Performance and optimization

Dedicated compilers for DSLs, like yacc, are often developed not because it is impossible to represent domain specific concepts in a general-purpose language, but because such a representation would have unacceptable overhead.

A good example are matrix libraries, which are heavily used in scientific computation. It is easy to implement matrices as a simple class library, but the overhead of doing so is unacceptably high, because calling a method is orders of magnitude slower than directly accessing an element of an array. Method inlining can eliminate this overhead to some extent, but inlining is not strong enough to deal with a second problem, which is memory use. A naive implementation of matrix operators has a high memory overhead, because it creates unnecessary intermediate data structures.

Consider the equation $E = A * B + C * D$, where A, B, C, and D are matrices, and + and $*$ are defined as element-wise addition and multiplication. A naive implemention of + and $*$ might define them as ordinary functions that create and return matrices. However, doing so would cause the equation to allocate three new arrays in memory to store intermediate values. A much more efficient way of performing the computation is to use *loop fusion*, as illustrated by the following for-loop. (We assume that each matrix has a data member, which is an array that holds the matrix data.)

```
for (int i = 0; i < A.data.length; i++) {
  E.data[i] = A.data[i] * B.data[i] + C.data[i] * D.data[i];
}
```

Performing loop fusion by hand is not only more verbose, it bypasses the layer of abstraction that the matrix library is supposed to provide. As a result, it does not work for matrices that use a non-standard memory layout, such as sparse arrays or array slices. What's more, it is not even as fast as it could be. It is often more efficient to break large arrays into tiles for better cache coherency [Veldhuizen, 1998], but tiles are also a non-standard memory layout. The choice of which memory layout to use, and the best way to access elements within that layout, should be made by the implementor of the matrix library, not the clients of the matrix library.

The Blitz++ array library demonstrates that the twin goals of having a good high-level interface, and having a high-performance implementation, do not have to be mutually exclusive [Veldhuizen, 1998]. By using a combination of inlining and C++

template meta-programming, the Blitz++ library not only eliminates the overhead that would ordinarily be associated with the use of classes and methods, but it performs various domain-specific optimizations, such as tiling, slicing, and loop-fusion. Blitz++ does not perform these optimizations in spite of the high-level interface, it is able to do them *because* it has a high-level interface.

1.3.2 Partial evaluation

Partial evaluation is a more principled way of performing the same kinds of optimizations that C++ templates allow. (Section 5.8 discusses why it is more principled.) Partial evaluation shifts computations from run-time to compile-time. If the data required by a computation is available at compile-time, then the computation can be performed at compile-time, and thus incurs no run-time penalty.

When implementing an interpreter for a domain-specific language, the major source of overhead is the fact that the interpreter must traverse the abstract syntax tree of the DSL, and branch on every node. However, the abstract syntax tree is often known at compile-time, so those branches can be eliminated. The following example makes use of the simple interpreter in Figure 1.2.

```
foo :: Int -> Int -> Int
foo x y = eval (Plus (Lit x) (Lit y))
-- partially evaluated to:  foo x y = x + y
```

In this example, x and y are variables, the values of which are not statically known (i.e. known at compile-time). However, the abstract syntax tree is statically known, so the eval function can be executed at compile-time. Shifting the computation from compile-time to run-time results in a very large speedup in this case; a single addition operation is orders of magnitude faster than constructing an AST, traversing the AST, branching on each node, and then garbage-collecting the result.

In practical applications, this difference in efficiency is often the difference between using a high-level library, and writing out optimized code by hand, so the potential gains in programmer productivity can be quite large.

1.3.3 The tag-elimination problem

The tag elimination problem was identified by Jones in [Jones et al., 1993] as an outstanding problem in the field of partial evaluation for typed languages, and has since been discussed by many authors [Hughes, 1996] [Taha and Makholm, 2000] [Pašalić

```
data Univ = IntV Int
          | FunV (Univ -> Univ)

data Expr = Lit  Int        -- integer literal
          | Var  Int        -- x   (using De Bruijn indices)
          | Fun  Expr       -- λx. t
          | Plus Expr Expr  -- t + u
          | App  Expr Expr  -- t(u)

-- evaluate expression with respect to environment
eval :: [Univ] -> Expr -> Univ
eval env (Lit i)    = IntV i
eval env (Var i)    = env !! i
eval env (Fun b)    = FunV (\x -> eval (x:env) b)
eval env (Plus a b) = uadd (eval env a) (eval env b)
eval env (App f a)  = uapp (eval env f) (eval env a)

-- add two universal values; requires dynamic check
uadd :: Univ -> Univ -> Univ
uadd (IntV a) (IntV b) = IntV (a + b)
uadd _        _        = undefined

-- apply a universal value to an argument; requires dynamic check
uapp :: Univ -> Univ -> Univ
uapp (FunV f) a        = f(a)
uapp _        _        = undefined

-- timesTwo = eval [] (\x. x + x)
-- partially evaluates to:  FunV (\x:Univ. uadd x x)
timesTwo = eval [] (Fun (Plus (Var 0) (Var 0)))
```

Figure 1.5: A Haskell intpreter for the λ-calculus

et al., 2002] [Carette et al., 2007]. In a statically typed language, there are cases where a partial evaluator is not able to remove all of the interpretive overhead. To demonstrate the problem, we extend our simple interpreter so that it handles the λ-calculus, as shown in Figure 1.5.

The new version of the interpreter must deal with both integer values and function values, and it does so by storing all values in the universal data type Univ. As a result of this change, the interpreter must check when adding two values together that the arguments are, in fact, integers. The undefined keyword in Haskell will throw a run-time error if the arguments fail this check. Partial evaluation can eliminate the abstract syntax tree just as before, but it cannot eliminate the dynamic type checks. Instead of reducing the expression to x+x, it reduces the expression to uadd x x, which is much

slower.

This issue is not fundamentally a problem with the partial evaluator, it is a problem with the way that the interpreter is written. The interpreter in Figure 1.5 interprets expressions in the untyped λ-calculus. Thus, the expressions that it interprets are not necessarily well-typed. Any language that does not perform static type checking must instead perform dynamic type-checking, and that is exactly what has happened here.

1.3.4 Well-typed interpreters

In order to solve the tag elimination problem, we must implement a typed λ-calculus, such as the simply-typed λ-calculus, rather than the untyped λ-calculus. However, the solution involves more than simply doing a quick type-checking pass. We must prove that the evaluation of a well-typed expression in the DSL will also be a well-typed expression in the host language. Moreover, this proof must be be encoded within the type system of the host language.

Two general techniques have been proposed in the literature for dealing with this problem. First, generalized abstract data types (GADTs) [Peyton Jones et al., 2006] can be used to constrain the Expr data type, and ensure that only well-typed terms can be constructed. Alternatively, dependent types can be used to encode a proof of well-typedness for a given expression [Pašalić et al., 2002] [Augustsson and Carlsson, 1999]. With dependent types, the Expr data type remains the same, but the eval function uses the proof of well-typedness to eliminate dynamic casts.

1.4 Putting it together: the DSL expression problem

The "DSL expression problem" is a straightforward combination of both the expression problem and the tag elimination problem. A solution to the DSL expression problem consists of (1) a language L with its associated type system, (2) a partial evaluator for L, and (3) an interpreter written in L, which interprets the expressions of a simple *typed* DSL.

The solution must demonstrate that (a) it is possible to extend the interpreter with new data type variants, (b) that it is possible to extend the interpreter with new operations, and (c) that partial evaluation of the interpreter will eliminate all interpretive overhead. Since the DSL is typed, there cannot be any dynamic type casts. In addition, each extension must be separately type-checked and separately compiled, and it

must be possible to combine different extensions together without losing the partial evaluation properties.

A solution to the DSL expression problem has important practical applications. Like Blitz++, it is an example of a self-optimizing library. However, it is also an extensible library; the concepts provided by the library can be extended with new variants and optimizations.

1.5 Pure subtype systems

The main technical contribution of this thesis is an idea that we call *pure subtype systems*. Pure subtype systems simultaneously increase the expressive power of the type theory, and reduce its compexity. This combination of power and simplicity is what allows us to address the type system requirements given in section 1.2.5.

Pure Subtype Systems are based on two "big ideas", both of which are a radical departure from usual practice.

(1) The theory does not distinguish between types and objects.

(2) The theory is based on subtyping, rather than typing.

Before explaining these two ideas in more detail, it is important to clarify some terminology. The *terms* of a language include everything defined by the formal syntax of that language. Most type theories divide terms into two or three *sorts*, which are syntactically different. The two main sorts are *objects* and *types*; many systems also define *kinds*.

The word "object" here refers to the objects of the theory, not to "objects" in the sense of object-oriented programming. The word "term" refers to any valid piece of syntax. Some authors in the literature use the word "term" to denote objects, as distinct from types; our use of the word "term" includes both objects and types.

In addition to the difference between types and objects, the syntax of a language can be further divided into *values* and *expressions*. A value (e.g. a function) introduces an object. An expression (e.g. function application) is associated with a reduction rule that eliminates a value. Values are the end result of evaluation; if a term does not contain any free variables, then the result of evaluating that term will be a value (assuming that evaluation terminates).

1.6 Types and Objects

There are various reasons, both historical and technical, for distinguishing between objects and types. These reasons are discussed in Chapter 5. This section provides a few motivating reasons for why we might wish to treat them uniformly.

1.6.1 Symmetry

Simple type systems, such as the simply-typed λ-calculus or System F, only support *proper types*, which are values at the type level. A *proper type*, e.g. Int or Float → Float, is the type of an object. More sophisticated type systems like System F_ω include type operators, which are functions like List that map from types to types. They also include type expressions, e.g. List(Int), which can be evaluated.

Note that although they look similar, the type operators provided by System F_ω are stronger than the parameterized types (a.k.a. generics) found in Java 1.5. In Java 1.5, the type List<Int> is essentially treated as a proper type, not an expression that is evaluated. System F_ω supports true computations at the type level, including higher-order functions over types.

In System F_ω, operators and expressions at the type level exactly mirror functions and expressions at the object level. The λ-calculus provides an effective model of computation for both levels. Extrapolating further, one might hypothesize that *any* construct which is useful for computation at the object level might also be useful at the type level. This observation is hardly new; it was formalized some time ago in the context of Pure Type Systems, which are one of the cornerstones of modern type theory [Barendregt, 1992].

Pure Type Systems unify types and objects to a partial degree by exploiting what we call "the symmetry of expressions". Expressions in this case are terms, such as functions and function application, that are associated with reduction rules. Since it is possible to define functions and function application for objects, it is also possible to define them for types; the basic reduction rules in a Pure Type System are the same at both the type and object levels.

However, Pure Type Systems do not completely unify types and objects because they still maintain a distinction between levels. Moreover, Pure Type Systems fail to exploit an additional symmetry, which we call "the symmetry of values".

1.6.1.1 The symmetry of values

There is a pattern which appears in the surface syntax of a number of different type systems. Every distinct syntax for values at the object level has a corresponding syntax for proper types at the type level, and vice versa. In other words, *types describe the shape of values*.

The following table illustrates this symmetry for several common types. The meta-variables s,t,u denote objects, S,T,U denote types, while l denotes a label. The notation $\bar{t}$ denotes a (possibly empty) sequence of objects $t_1..t_n$, similarly for types and labels.

description	value	proper type
scalar value	a	A
function	$\lambda x : T.\ u$	$T \to U$ (simple) or $\Pi x : T.U$ (dependent)
pair	(t,u)	(T,U)
record	$\{\bar{l} = \bar{t}\}$	$\{\bar{l} : \overline{T}\}$

Notice that not only does each kind of value correspond to a particular type, but the syntax for types and values is remarkably similar. Despite this similarity, most type theories, including Pure Type Systems, distinguish between the two.

Not only do objects values and proper types have a different syntax, they have a different semantics. Every object value is paired with an elimination rule that allows the value to be used in computations. If f is a function, then $f(a)$ is an expression which applies it to the argument a. Similarly, if r is a record, then $r.l$ is an expression that projects the slot l from the record. Proper types do not have computational behavior; it is not possible to apply a Π-type, or project a slot from a record type.

Some recent work has demonstrated that the distinction between λ and Π is somewhat artificial and can be safely eliminated [Kamareddine, 2005]. There are also practical reasons for wishing to discard it [Peyton Jones and Meijer, 1997]. Pure subtype systems follow this trend and use the same syntax for both values and proper types. Moreover, in pure subtype systems, types do have computational behavior. If f is a function which has type F, then F is also a function with the following property: for any valid argument a, $F(a)$ will return the type of $f(a)$. Similarly for records: if $r : R$, then $r.l : R.l$.

1.6.2 Dependent types

Dependent types are types which are parameterized by objects. A classic example is the type Array(T, n): an array of type T and size n, where n is a natural number.

Although dependent types have been widely used in theorem provers, they are seldom found in general-purpose languages, except in a very restricted form. Full dependent types are problematic because the complexity of the object language gets carried over to the type language. Consider the following questions, which a type-checker might be asked to answer:

Does	Array(T, 10)	=	Array(T, 5 + 5)?
Does	Array(T, 10)	=	Array(T, sqrt(100))?
Does	Array(T, x)	=	Array(T, $(\lambda y.\ y)(x)$) for any x?
Does	Array(T, x+y)	=	Array(T, y+x) for any x, y?

All of these questions involve reasoning about object expressions. A good C++ compiler can handle the first case because it evaluates constant arithmetic expressions at compile time, but it will not handle any of the other cases. The second case requires the type checker to evaluate an arbitrary function with constant arguments. The third case requires partial evaluation of a function with unknown arguments. The fourth case requires the type-checker to know basic identities of addition, in this case commutativity.

In general, dependent type systems must set a practical limit regarding how much reasoning, and what kind of reasoning, they are willing to do. The theory of pure subtype systems that we present here uses intensional equality for terms, which means that terms are equal only if they evaluate to a common result. Pure subtype systems can thus handle the first three cases, but not the fourth.

The main advantage of distinguishing between types and objects is that the language of types can be simpler than the language of objects. Most importantly, the language of types can be strongly normalizing, even if the language of objects is not; strong normalization for types is necessary for decidable type checking. Dependent types remove this advantage, because they allow types and objects to be freely mixed.

If types and objects can be freely mixed, then there is no longer a compelling reason to distinguish between the two. Chapter 5 discusses the trade-off in more detail.

1.6.3 Modules

The argument based on symmetry is aesthetic rather than practical, and the argument based on dependent types boils down to "why not?" rather than "why should we?". The main practical motivation for unifying types and objects is that it makes it easier to deal with modules. The full semantics of modules are the subject of later chapters; what follows is an informal discussion.

A module is a record which contains named declarations. The declarations in a module may refer to one another, and they may declare either types or objects. Consider the following pseduo-code:

```
module m = {
  type T       = ...;
  foo(x: T): T = ...;
};

twicefoo(x: m.T): m.T = m.foo(m.foo(x));
```

The module m has two members: T is a type, while foo is an object. The function twicefoo uses both members. This declaration raises an important question: is module m a type or an object? It's hard to say, since it contains both type and object members.

One approach is to treat m as an existential sum — an object parameterized by a hidden type. An existential is appropriate in this case because foo depends on T, but not the other way around. Standard or "weak" existentials don't quite work, because the type is not actually hidden; the expression m.T projects T from m. Nevertheless "strong" existential sums come equipped with such projections [Luo, 1994]. However, in the presence of dependent types and recursive modules, we could easily have a type declaration and an object declaration that are mutually dependent, with no clear way to separate the two.

Another approach, used by Standard ML, is to treat modules as a third sort of term: neither type nor object. Modules in ML are not first-class objects; they are given a distinct semantics [Harper and Stone, 2000]. All of the standard operations on types and objects are duplicated yet again in the module system; ML functors are functions over modules, ML signatures are the types of modules, and so on.

This thesis presents an alternative approach. We argue that *it does not matter whether a module is a type or an object.* There is a single syntax for terms, a single mechanism for quantifying over terms, and the same syntax can be used for objects, types, kinds, modules, classes, prototypes (see below), or anything else.

1.6.4 Classes and Prototypes

Classes in OO languages are similar to modules in many ways, and thus present the many of the same difficulties. A class *C* serves three main roles:

1. It is a type for instances of *C*.
2. It provides constructors for creating instances of *C*.
3. It defines methods for operating on instances of *C*.

Methods and constructors are functions (i.e. objects), so a class, like a module, combines types and objects into a single structure. Note that in a functional language, the above three roles are distinctly different, and would be handled by different sorts of term. For example, in Haskell and ML List is a type, while the contructors for lists (i.e. cons and nil) are objects. Operations on lists are also objects, and would be defined as part of a type class in Haskell, or a module in ML.

The fact that classes serve multiple roles can sometimes cause confusion. For example, Java generics allow classes to be used as type parameters. However, a type parameter *X* is just a type, not a class, and it therefore has a number of restrictions when compared to "real" classes:

- It is not possible to call constructors, e.g. new *X*().
- It is not possible to use run-time type information, e.g. *a* instanceof *X*.
- It is not possible to use type casts, e.g. (*X*) expr.
- It is not possible to use reflection, e.g. *X*.class.

These restrictions are a consequence of the fact that Java generics are implemented with *type erasure*, which removes type parameters from the compiled code. Type erasure is one of the hallmarks of a strong *phase distinction* [Harper et al., 1989] between types and objects (see Section 5.2.1.3 for further discussion). Type erasure in Java causes confusion because the rest of Java does not have a strong phase distinction. True classes are not erased in this way; they are reified as objects at run-time, and they can be used at run-time in various ways.

The principle of classes-as-objects is even more pronounced in languages like smalltalk, and it reaches its logical conclusion in *prototype* languages, most notably Self [Ungar and Smith, 1987]. In Self, both classes and instances are ordinary objects. A "class" is a repository for methods which are shared between instances, and each instance *delegates* certain behavior to its "class". In Self, "subclasses" delegate to "su-

perclasses" in exactly the same way that instances delegate to "classes". Delegation in Self is thus quite different from inheritance in class-based languages like Java, because in Java, inheritance and instantation are two very different operations.

In formal type systems for class-based languages, instantiation is modeled by typing (e.g. $t : T$), while inheritance is modeled by subtyping (e.g. $T \leq U$). Self is not statically typed, so there is no formal type system for it. However, if there were such a system, it would have to combine the notions of typing and subtyping. This thesis presents the DEEP calculus, which is a type theory that combines typing and subtyping in exactly this way. In many respects, DEEP can be seen as a foundational calculus for prototype languages like Self.

1.7 Subtyping

In practical programming, there is often a need to quantify over a set of types which have some common property. For example, the min function takes the minimum of two objects. It can be defined as a polymorphic function that ranges all proper types which have an order "$<$" defined on them, e.g.

$$\mathsf{min}(T \leq \mathsf{Ordered},\ x : T,\ y : T)\ =\ \mathsf{if}\ (x < y)\ \mathsf{then}\ x\ \mathsf{else}\ y$$

Standard typing (and kinding) is too coarse-grained to capture definitions like the one above. The kind $*$ quantifies over all proper types; it is difficult to distinguish a particular subset using kinds alone. Subtyping is one way of distinguishing between types at a finer level of granularity. There are other ways, most notably type classes [Wadler and Blott, 1989] [Peyton Jones et al., 1997] and qualified types [Jones, 1992], but subtyping is a good fit with the inheritance and delegation mechanisms that we are interested in applying to modules.

The subtype relation organizes all types into a classification hierarchy, much like the Linnaean taxonomy used in biology. Subtyping has received a lot of attention in the literature as the basis for OO inheritance. The popularity of OO languages can be partly attributed to the fact that classification hierarchies are widely used outside of computer science, and are thus familiar and intuitive to most programmers.

1.7.1 *Subtyping*: a *sub*stitute for *typing*

In a traditional type system, typing is regarded as the primary relation between terms, while subtyping is a refinement. Subtyping is secondary because it can be removed from the type system without breaking any fundamental properties. A type system without subtyping may be less expressive than one with subtyping, but it is still safe.

This thesis presents a new approach type theory that we call pure subtype systems. Pure subtype systems break with tradition because they take subtyping, rather than typing, to be the primary relation between terms. In fact, there is no typing relation at all; typing is shown to be a special case of subtyping.

In pure subtype systems, the subtype relation is defined over all terms, not just types. In the case of functions, subtyping is point-wise. Given two functions f and g, f is a subtype of g if and only if $f(a)$ is a subtype of $g(a)$ for any valid argument a.

This definition of point-wise subtyping is not new. The idea first appeared in System $F^{\omega}_{\leq}$, where it is called *higher-order subtyping*. System $F^{\omega}_{\leq}$ extends the subtype relation from proper types to *higher-order types*. Higher order types are another name for type operators, i.e. functions like List that map from types to types.

Our definition of point-wise subtyping has been lifted verbatim from System $F^{\omega}_{\leq}$. The only change that we have made is to extend the pointwise subtyping rule so that it covers all functions, not just type operators. Since every object in a pure type system can be interpreted as a type, every function can be interpreted as a type operator.

1.7.1.1 Removing the barrier between types and objects

Figure 1.6 illustrates the relationship between objects, types, and kinds, and between typing and subtyping in System $F^{\omega}_{\leq}$, and compares it to pure subtype systems. (Note that the diagram is intended as an illustration only; the particular subtype relationships depend on the precise way in which types like Int, String and Ordered are encoded. In particular, there are good reasons why Eq and Ordered should not be considered to be supertypes of Int and String, but that's not the point of the diagram.)

Typing is a relationship between terms at different levels – between objects and types, and between types and kinds. Subtyping is a relationship between terms at the same level, but is very similar to typing in most other respects.

The hierarchy for pure subtype sytems differs from $F^{\omega}_{\leq}$ as follows:

1. Subtyping is defined over all terms.
2. There is no distinction between objects, types, and kinds.

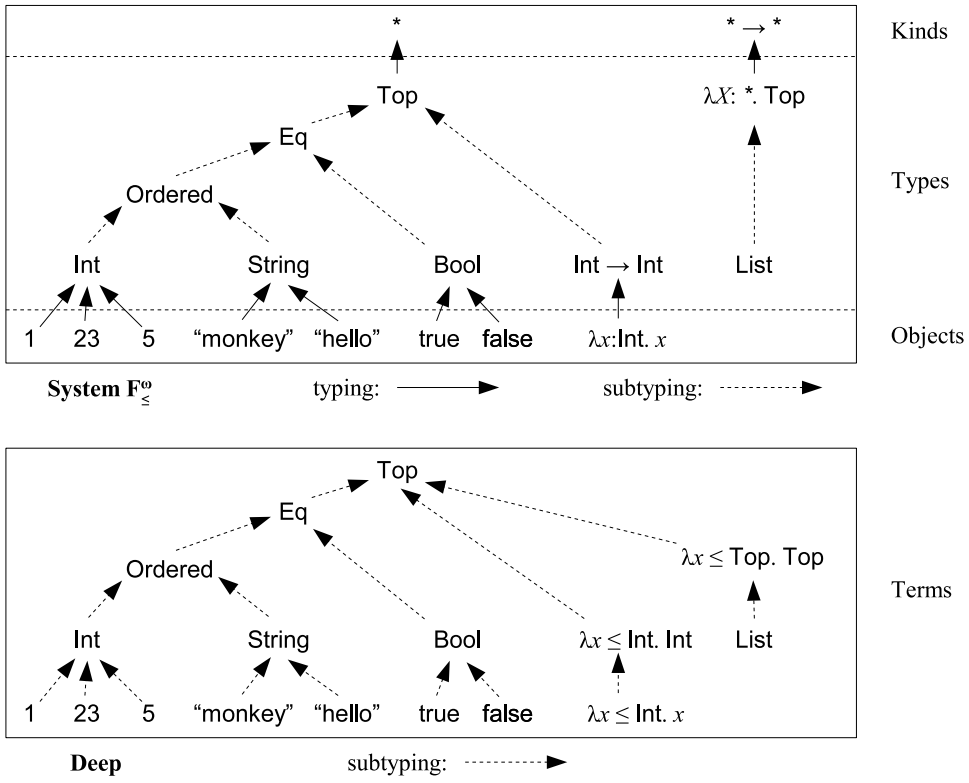


Figure 1.6: The type and subtype hierarchy in Systems $F^{\omega}_{\leq}$ and DEEP.

3. Every type relationship has been replaced with a subtype relationship; objects are subtypes of types.
4. The type Int → Int is now a function: $\lambda x \leq$ Int. Int.
5. Kinds disappear; their role is subsumed by Top-types.
6. The supertype Top quantifies over all terms, not just proper types.

Only one of these changes affects the overall organization of the hierarchy: the fact that Top quantifies over all terms. This change makes the type system fully impredicative, which has certain consequences for consistency and decidability. Impredicativity will be discussed in the next chapter.

1.7.2 Interpretation of types and objects

In the pure subtype systems, every term can be interpreted as either a type or an object, depending on context. A concrete object, like the number 3, is treated as a *singleton type* – it represents the type of all integers that are equal to 3.

Similarly, every type can be interpreted as an object. For example, let f be a

function which has type F. Then F is also a function with the following property: for any valid argument a, $F(a)$ will return the type of $f(a)$. This property results from the fact that typing is a special case of subtyping, and subtyping between functions is point-wise. The relationship can be clearly seen in the interpretation of arrow types shown above: the type $\mathsf{Int} \rightarrow \mathsf{Int}$ is a function: $\lambda x \leq \mathsf{Int}.\ \mathsf{Int}$.

This interpretation of types is very similar to *abstract interpretation* (see Section 2.8.1 for details). A type is regarded as an approximation of an object; it represents an object whose value is unknown. It is possible to perform computations with types, but the result will also be an approximation, i.e. a type. In a pure subtype systems, the type of an expression is calculated by substituting types for unknown variables in the expression, and then evaluating the expression, which yields a type as the result.

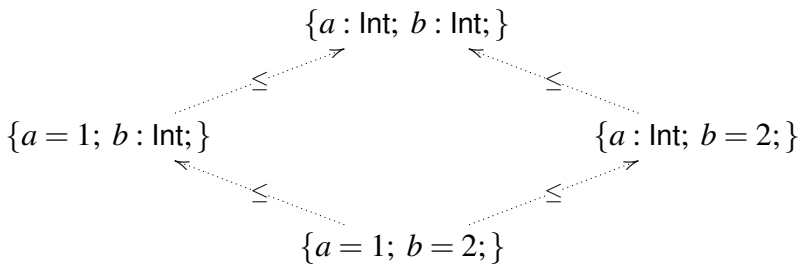
1.7.3 A spectrum of type information

If the subtype relation is extended to include all terms, then subtyping provides a smooth spectrum of type information which extends from singleton types, which provide total information about an object, all the way up to Top, which provides no information about an object. This spectrum means that the type system can make finer distinctions than would otherwise be possible.

For example, a typical typing rule for records would make the following judgement:

$$\{a = 1;\ b = 2;\} \ :\ \{a : \mathsf{Int};\ b : \mathsf{Int};\}$$

With ordinary typing, there are no intermediate terms between the object on the left, and the type on the right. Either a and b are both known, or they are both unknown. With subtyping, it is possible to write the following, where a is known, and b is not:



This distinction is of more than academic interest. Java allows certain methods to be declared *final*, which means that they cannot be overridden in subclasses. Compilers routinely inline final methods on the basis of type information; it's an important

optimization for OO programs. The middle types correspond to classes in which *a* and *b* are final.

1.7.4 Combining typing and partial evaluation

One of the most interesting things about the subtype relation is the fact that it essentially combines typing and partial evaluation into a single judgement.

As explained in Section 1.6.2, type equality in a dependent type system involves object equality, and object equality requires partial evaluation of object expressions. Thus, partial evaluation can be seen as an intrinsic part of typing. In many cases, reducing an object expression will also change its type; the type becomes more specific as reduction proceeds. Partial evaluation can make certain programs typable that would not be typable otherwise. Moreover, as explained above, partial evaluation can benefit from the use of type information. Types allow reductions, such as inlining of final bindings, that would not be possible in an untyped setting.

One of the practical benefits of our approach is the fact that in a pure subtype system, the partial evaluator and the type system are closely coupled together; each part can make use of the information provided by the other.

1.8 Outline of thesis

This thesis can be broadly divided into two halves. Both halves are concerned with pure subtype systems, but they represent separate lines of research that can be understood independently. The reader can safely read one half without the other.

The first half of the thesis presents the theory of pure subtype systems in the form of a typed λ-calculus. By presenting it in this form, we are able to study the basic theory and metatheory in detail, and compare and contrast pure subtype systems with more traditional approaches to type theory. Chapter 2 contains this material.

The second half of the thesis constructs a theory of modules that is based on pure subtype systems. The second half is focused primarily on practical programming; we show how virtual classes and deep mixin composition can be encoded, and present our solution to the DSL expression problem. Chapters 3 and 4 contain this material.

Overall, we present the material gradually in the form of three calculi: System $\lambda_{\triangleleft}$, DEEP--, and DEEP, where each calculus builds on the ones before it.

1.8.1 Outline of individual chapters

The chapters of this thesis are organized as follows:

Chapter 1 (this chapter) introduces the problems we wish to solve, and provides an informal description of how we intend to go about solving them.

Chapter 2 presents System $\lambda_{\triangleleft}$, which is a typed λ-calculus. Although the calculus is simple, it has the two main features that distinguish pure subtype systems: it does not differentiate between types and objects, and it relies on subtyping rather than typing. We use System $\lambda_{\triangleleft}$ to develop the basic meta-theory behind pure subtype systems, so this chapter is the most technical chapter in the thesis. Unfortunately, we are not able to prove two of the most important properties, namely type safety and transitivity elimination, but we present a partial proof of these properties. In particular, we show that subtyping can be formulated as a reduction system, and that type safety and transitivity elimination follow from commutativity of the underlying reductions. We can prove that reductions commute locally, but we have so far been unable to prove that that they commute globally.

Chapter 3 presents the DEEP-- calculus, which extends System $\lambda_{\triangleleft}$ with support for modules, general recursion, and inheritance. We present the basic type theory for DEEP--, and give a number of examples of how the theory can be used in practice to solve various difficult typing problems that have been previously mentioned in the literature. The DEEP-- calculus is also integrated with a partial evaluator, and we discuss how subtyping and partial evaluation are related.

Chapter 4 presents the DEEP calculus, which is the most sophisticated calculus that we consider. The DEEP calculus extends DEEP-- with polarized higher-order subtyping [Steffen, 1997] and mixins. Both of these extensions are necessary to solve the DSL expression problem. We present two solutions to the expression problem in DEEP: one which uses an object-oriented style, and one which uses a functional programming style. These two solutions correspond to the pseudo-code solutions given earlier in this chapter. We also present our solution to the DSL expression problem.

Chapter 5 gives an overview of related work in the literature. The purpose of chapter 5 is mainly to describe how pure type systems and DEEP relate to other research areas. Some parts of the literature that are directly related to the design of System $\lambda_{\triangleleft}$ and the DEEP-- and DEEP calculi are discussed in chapters 2-4 rather than here.

Chapter 6 offers a few concluding notes.

Chapter 2

Pure Subtype Systems

Alternate title: "How to eat a pepper grinder, with proofs."

> *Now, be sure not to confuse the roles of a class C, and an object O belonging to class C. To give a really shaky analogy, a class is like a pepper grinder, and an object is like pepper. You can use the pepper grinder (the class) to create pepper (objects of that class) by turning the crank (calling the Spawn function).* ***But*** *a pepper grinder... is not pepper...* ***so you must not try to eat it!***
>
> — Tim Sweeny, UnrealScript documentation

2.1 Introduction

The λ-calculus, first introduced by Church in the 1930s, is by far the most widely studied formal model of programming languages [Church, 1940]. Many well-known type systems are built on the λ-calculus, so it provides a good framework in which to compare and contrast different systems. In keeping with tradition, the main ideas of this thesis are developed first for the λ-calculus. Later chapters then apply these ideas to various extensions, such as modules, classes, and inheritance.

This chapter introduces System $\lambda_{\triangleleft}$, which is a specific example of a larger class of calculi that we will refer to as *pure subtype systems*. Pure subtype systems have the following two characteristics, as discussed in Chapter 1:

(1) They do not distinguish between types and objects.

(2) They are based on subtyping, rather than typing.

2.1.0.1 Outline of chapter

This chapter is organized as follows.

- Section 2.2 provides an informal description of some of the basic concepts of System $\lambda_{\lhd}$, and compares it to pure type systems and System $F^{\omega}_{\leq}$.
- Section 2.3 introduces the formal syntax and semantics of System $\lambda_{\lhd}$, and provides examples of how subtyping in the system works.
- Section 2.4 shows how other type theories can be embedded in System $\lambda_{\lhd}$. We provide embeddings for System $F_{\leq}$ and System $\lambda*$, the pure type system with $* : *$. The embedding of System $\lambda*$ shows that System $\lambda_{\lhd}$ suffers from Girard's paradox and is thus not strongly normalizing.
- Section 2.5 establishes some basic meta-theoretic properties of System $\lambda_{\lhd}$, including a proof of type safety. Type safety depends on transitivity elimination, which we present as a conjecture.
- Section 2.6 introduces the algorithmic formulation of subtyping. In the algorithmic formulation, the subtype relation is treated as an abstract reduction system. Transitivity elimination follows from commutativity of the underlying reductions. We prove basic meta-theoretic properties of the algorithmic system, and show that the declarative formulation and the algorithmic formulations are equivalent.
- Section 2.7 discusses commutativity, which is the main meta-theoretical result for System $\lambda_{\lhd}$. Unfortunately, we are only able to prove that subtype reductions commute locally; we are not able to prove that they commute globally.
- Section 2.8 shows how subtype reduction can be used to construct practical algorithms for comparing terms and finding minimal supertypes. However, because System $\lambda_{\lhd}$ is not strongly normalizing, these algorithms are neither decidable nor complete.
- Section 2.9 Summarizes how our approach to subtyping is related to previous formulations in the literature, and discusses possible directions for future research.
- Section 2.10 wraps up by discussing how our results for System $\lambda_{\lhd}$ relate to type theory in general.

2.2 Comparison to System $F^{\omega}_{\leq}$ and Pure Type Systems

System $\lambda_{\triangleleft}$ can be best understood by comparison to two other well-known calculi: System $F^{\omega}_{\leq}$, and Pure type systems (PTSs).

Higher-order subtyping: System $F^{\omega}_{\leq}$ is a theory of higher-order subtyping [Steffen and Pierce, 1994] [Compagnoni and Goguen, 2003]. Higher-order types are functions, such as List, that map from types to types. Given any type T, List(T) is the type of lists which contain elements of type T. Such functions are also called type operators. *Higher-order subtyping* extends the subtype relation so that it includes higher-order types.

Simpler theories of subtyping, such as System $F_{\leq}$, only define the subtype relation over *proper types*, which are the types of objects. Each proper type denotes a set of objects. The subtype relation between proper types thus corresponds to a subset relation between sets; a type T is a subtype of U if and only if every object of type T is also an object of type U.

Higher order types do not denote sets of objects; they denote functions over sets. Thus, the idea that "subtypes are subsets" is no longer valid. The primary insight behind System $F^{\omega}_{\leq}$ is that it is possible to define a meaningful subtype relation directly over functions. System $F^{\omega}_{\leq}$ uses a pointwise subtyping rule: given two functions F and G, $F \leq G$ if and only if $F(A) \leq G(A)$ for any argument A.

The subtype relation for System $\lambda_{\triangleleft}$ has been lifted verbatim from System $F^{\omega}_{\leq}$. The only change that we have made is to extend the pointwise subtyping rule so that it covers all functions, not just type operators.

Pure type systems: Pure type systems are a class of λ-calculi which use a uniform syntax for expressions at both the object level and the type level [Barendregt, 1992]. By adopting a uniform syntax, they are able to exploit some of the symmetries that are found in more sophisticated type systems.

For example, System F_{ω} is the theory of higher-order types (without subtyping). In essence, System F_{ω} extends System F by including a complete copy of the simply-typed λ-calculus "one level up", so that functions can be defined over types as well as objects. The rules which assign kinds to functions over types are identical to the rules which assign types to functions over objects.

In the standard presentation of System F_{ω} [Pierce, 2002], functions over types have a different syntax from functions over objects. An function over objects is written

$\lambda x : T.\ t$, where T is a type, t is an object, and x is an object variable. A function over types is written $\lambda X : K.\ T$, where K is a kind, T is a type, and X is a type variable. These two functions are obviously similar, but they are syntactically distinct.

Rather than separating the set of terms into objects t, types T, and kinds K, pure type systems use a unified set of terms which ranges over objects, types, and kinds. This simplification leads to a very elegant and compact system, since the same inference rules can be used for both typing and kinding. Objects, types, and kinds still belong to different universes in a PTS, but the universe of a term is not specified syntactically, it is determined by the typing judgement. In fact, PTSs are flexible enough to model many different universe structures within the same formal framework.

Like pure type systems, System $\lambda_{\lhd}$ uses a uniform syntax for terms. However, instead of defining a type relation over terms, it defines a subtype relation over terms.

2.2.1 Deconstructing the typing judgment

System $F^{\omega}_{\leq}$ and Pure Type Systems are examples of what I will refer to as "traditional type systems", which maintain a strong distinction between types and objects. Typing (and kinding) in these systems is an all-purpose judgment which gives three distinct pieces of information:

1. The type of a term describes the *shape* of the value that will be produced when the term is evaluated.
2. Terms which can be assigned types are said to be *well-typed*. The evaluation of a well-typed term will not generate type errors at run-time.
3. In a PTS, the type or kind of a term describes the level or universe in which the term resides.

In system $\lambda_{\lhd}$, these three pieces of information are split into separate judgments:

1. The subtype relation compares the shapes of terms.
2. The well-formedness judgment determines whether a term is well-typed.
3. The universe judgement determines the universe in which the term resides; we regard universes as optional.

This design is motivated by the observation that typing and subtyping have a significant amount of overlap, because they both involve a judgment about shape. The judgment $t : U \to S$ says that the object t will produce a function when it is evaluated.

Similarly, the judgment $T \leq U \rightarrow S$ states that the type expression T will produce a function type when it is evaluated.

System $\lambda_{\triangleleft}$ eliminates overlap by removing the notion of "shape" from the typing judgment. The well-formedness judgment consists of the remaining, non-shape portions that are left behind. Because there is no typing judgment, types do not have a special syntax, and they are not given any special treatment. All terms can act as either types or objects depending on context.

System $\lambda_{\triangleleft}$ uses a single universe for both types and objects, so a universe judgement is not required. However, we will show that adding a universe judgement restores the familiar level stratification that is present in traditional type systems, without disturbing the other meta-theoretic properties of the type system.

2.2.2 Bounded quantification and Top-types

The elimination of types from the syntax of System $\lambda_{\triangleleft}$, and the replacement of typing with subtyping has a precedent in the literature. It corresponds exactly to the elimination of kinds, and the replacement of kinding with subtyping in System $F^{\omega}_{\leq}$. We can see the elimination of kinds by comparing System $F^{\omega}_{\leq}$ to System F_{ω}.

Systems F_{ω} and $F^{\omega}_{\leq}$ define no less than less than six pieces of syntax for λ-abstractions and proper types. In the following table, s, t, u are objects, S, T, U range over types, and K_1, K_2 range over kinds; x is an object variable, while X is a type variable. For the sake of consistency with Pure Type Systems, we use a slight change of notation from Pierce [Pierce, 2002], and adopt Π instead of $\forall$ as the concrete syntax for proper types.

description		abstraction	proper type or kind
ordinary functions	(both)	$\lambda x : T.\ u$	$T \rightarrow U$
polymorphic objects	(F_{ω})	$\lambda X : K.\ u$	$\Pi X : K.U$
polymorphic objects	($F^{\omega}_{\leq}$)	$\lambda X \leq T.\ u$	$\Pi X \leq T.\ U$
type operators	(F_{ω})	$\lambda X : K.\ U$	$K_1 \rightarrow K_2$
type operators	($F^{\omega}_{\leq}$)	$\lambda X \leq T.\ U$	$\Pi X \leq T.\ K$

Type abstractions in System F_{ω} use *kinding*, which has the syntax $\lambda X : K.\ U$. Type abstractions in System $F^{\omega}_{\leq}$ use *bounded quantification*, which has the syntax $\lambda X \leq T.\ U$. This substitution is possible because subtyping in System $F^{\omega}_{\leq}$ is strictly more expressive than kinding.

The language of kinds in System F_ω is defined as: $K ::= * \mid K \to K$. Every kind is associated with a Top-type, the supertype of all types which have that particular kind. The Top-type of kind K, written $\mathrm{top}(K)$, is:

$$\begin{array}{lcl} \mathrm{top}(*) & = & \mathsf{Top} \\ \mathrm{top}(K_1 \to K_2) & = & \lambda X \leq \mathrm{top}(K_1).\ \mathrm{top}(K_2), \text{ where } X \text{ chosen fresh.} \end{array}$$

Thus, every abstraction of the form $\lambda X : K.\ U$ can be replaced with an abstraction of the form $\lambda X \leq T.\ U$ without loss of generality. This replacement is interesting, because λ-abstractions are the only place in the syntax where kinds actually appear. Replacing kinding with bounded quantification causes kinds to disappear from the syntax of types, although they are still used internally to ensure that types are well-kinded.

2.2.2.1 Variants of System $F^\omega_\leq$

As a historical note, there are actually two variants of System $F^\omega_\leq$. The first variant, formalized by Steffen and Pierce, uses bounded quantification only on polymorphic objects [Steffen and Pierce, 1994]. The second variant, formalized by Compagnoni and Goguen, extends bounded quantification to type operators as well [Compagnoni and Goguen, 2003]. Kinds only disappear from the syntax of the second variant. The meta-theory for the second variant is considerably more complex than the first, for reasons that are discussed in Section 2.9.2.

2.2.2.2 Pure Subtype Systems

System F_ω is a member of the λ-cube [Barendregt, 1992], and thus can be encoded as a pure type system. Pure type systems have a much simpler syntax; the three different λ-abstractions become a single λ-abstraction, and the three different proper types/kinds become a single Π-type.

Unfortunately, it is not possible to combine the three λ-abstractions in System $F^\omega_\leq$ together without serious changes, because they are based on different judgments. Ordinary functions use typing, e.g. $\lambda x : T.\ u$, while polymorphic objects and type operators use bounded quantification, e.g. $\lambda X \leq T.\ U$.

System $\lambda_\lhd$ is a "pure subtype system." It does the same thing for System $F^\omega_\leq$ that pure type systems do for System F_ω. We resolve the discrepancy between typing and bounded quantification by extending the subtype relation so that it covers all terms,

Syntax:

x, y, z	Variable		
$s, t, u ::=$	Terms	$\Gamma ::=$	Contexts
x	variable	$\emptyset$	empty context
Top	universal supertype	$\Gamma, x \leq t$	variable
$\lambda x \leq t.\ u$	function		
$t(u)$	application	$\lhd ::=$	Type relations
		$\leq$	subtype
$v, w ::=$	Values	$\equiv$	type equivalence
Top	universal supertype		
$\lambda x \leq t.\ u$	function		

Context:
$C ::= [] \mid C(t) \mid t(C) \mid \lambda x \leq C.\ t \mid \lambda x \leq t.\ C$

Operational Semantics (Reduction): $\boxed{t \longrightarrow t'}$

$$\frac{t \longrightarrow t'}{C[t] \longrightarrow C[t']} \quad \text{(E-CONG)}$$

$$(\lambda x \leq t.\ u)(s) \longrightarrow [x \mapsto s]u \quad \text{(E-APP)}$$

Notation:

- $C[t]$ represents a context C with the term t substituted for the hole [] in the context.
- $[x \mapsto t]u$ denotes the capture-avoiding substitution of the term t for the variable x within u.

Figure 2.1: System $\lambda_{\lhd}$ — Syntax and Operational Semantics

---

including objects as well as types. Typing is replaced with bounded quantification in all cases, which allows the syntax to be unified. Abstractions of the form $\lambda x : T.\ u$ become abstractions of the form $\lambda x \leq t.\ u$. However, as a result of this change, proper types disappear from the syntax of terms, just as kinds disappear from the syntax of types in System $F^{\omega}_{\leq}$.

2.3 System $\lambda_{\lhd}$

The syntax and operational semantics of System $\lambda_{\lhd}$ are shown in figure 2.1.

The design of system $\lambda_{\lhd}$ has been heavily influenced by System $F^{\omega}_{\leq}$. In fact, System $\lambda_{\lhd}$ is essentially a fragment of $F^{\omega}_{\leq}$, a fragment that only contains type operators. Because every term in System $\lambda_{\lhd}$ can be interpreted as a type, every function can be

Subtyping: $\boxed{\Gamma \vdash t \lhd u}$

$$\frac{\Gamma \vdash s \lhd t, \quad t \lhd u, \quad t \text{ wf}}{\Gamma \vdash s \lhd u} \text{ (DS-TRANS)}$$

$$\frac{\Gamma \vdash u \equiv t}{\Gamma \vdash t \equiv u} \text{ (DS-SYM)} \qquad \frac{\Gamma \vdash t \equiv u}{\Gamma \vdash t \leq u} \text{ (DS-EQ)}$$

$$\Gamma \vdash x \equiv x \text{ (DS-VAR)}$$

$$\Gamma \vdash \mathsf{Top} \equiv \mathsf{Top} \text{ (DS-TOP)}$$

$$\frac{\Gamma \vdash t \equiv t' \quad \Gamma, x \leq t \vdash u \lhd u'}{\Gamma \vdash \lambda x \leq t.\, u \lhd \lambda x \leq t'.\, u'} \text{ (DS-FUN)}$$

$$\frac{\Gamma \vdash t \lhd t', \quad u \equiv u'}{\Gamma \vdash t(u) \lhd t'(u')} \text{ (DS-APP)}$$

$$\Gamma \vdash t \leq \mathsf{Top} \text{ (DS-ETOP)}$$

$$\frac{x \leq t \in \Gamma}{\Gamma \vdash x \leq t} \text{ (DS-EVAR)}$$

$$\Gamma \vdash (\lambda x \leq t.\, u)(s) \equiv [x \mapsto s]u \text{ (DS-EAPP)}$$

Well-subtyping: $\boxed{\Gamma \vdash t \lhd_{\text{wf}} u}$

$$\frac{\Gamma \vdash t \text{ wf}, \quad u \text{ wf}, \quad t \lhd u}{\Gamma \vdash t \lhd_{\text{wf}} u} \text{ (W-SUB)}$$

Context well-formedness: $\boxed{\Gamma \text{ wf}}$

$$\emptyset \text{ wf} \text{ (W-GAM1)}$$

$$\frac{\Gamma \text{ wf} \quad x \notin \text{dom}(\Gamma) \quad \Gamma \vdash t \text{ wf}}{\Gamma, x \leq t \text{ wf}} \text{ (W-GAM2)}$$

Term well-formedness: $\boxed{t \text{ wf}}$

$$\frac{\Gamma \text{ wf} \quad x \in \text{dom}(\Gamma)}{\Gamma \vdash x \text{ wf}} \text{ (W-VAR)}$$

$$\frac{\Gamma \text{ wf}}{\Gamma \vdash \mathsf{Top} \text{ wf}} \text{ (W-TOP)}$$

$$\frac{\Gamma, x \leq t \vdash u \text{ wf}}{\Gamma \vdash \lambda x \leq t.\, u \text{ wf}} \text{ (W-FUN)}$$

$$\frac{\Gamma \vdash t \leq_{\text{wf}} \lambda x \leq s.\, \mathsf{Top} \quad \Gamma \vdash u \leq_{\text{wf}} s}{\Gamma \vdash t(u) \text{ wf}} \text{ (W-APP)}$$

Notation:

- $\text{fv}(t)$ denotes the set of free variables in the term t.
- $\text{dom}(\Gamma)$ denotes the set of variables defined in Γ.
- $x \leq t \in \Gamma$ is true if the context Γ contains $x \leq t$.

We identify terms which are α-equivalent under renaming of bound variables. As usual, we assume that variables can be renamed as needed to avoid conflicts.
For compactness, we adopt the following convention: a pair of judgements $\Gamma \vdash J_1$ and $\Gamma \vdash J_2$ which are both made within the same context Γ are written as $\Gamma \vdash J_1, J_2$.

Note that $\lhd$ is a meta-variable which ranges over $\leq$ and $\equiv$.

Figure 2.2: System $\lambda_{\lhd}$— Declarative subtyping and well-formedness

.

interpreted as a type operator.

Functions are written: $\lambda x \leq t.\, u$. Note that both the type bound t and and the function body u are ordinary terms. Note also that functions use bounded quantification rather than typing; the above function can be applied to any term that is a subtype of t.

Top is a universal supertype which ranges over all terms; every term t is a subtype of Top. This interpretation of Top differs from System $F^{\omega}_{\leq}$. In System $F^{\omega}_{\leq}$, Top only ranges over the proper types (the types of objects), not over type operators. As is discussed in Section 2.4.5, the presence of Top make System $\lambda_{\triangleleft}$ fully impredicative, which has important consequences for decidability.

Unlike pure type systems, the dependent type $\Pi x : t.\ u$ is not a part of the syntax; this type is subsumed by ordinary functions. The arrow type $t \to u$ is not present either, for the same reason. We treat $t \to u$ as syntax sugar for an ordinary function $\lambda x \leq t.\ u$, where x is chosen fresh and does not appear in u.

2.3.1 Subtyping

The subtyping and well-formedness judgments are shown in Figure 2.2. This is the declarative formulation of subtyping (as distinct from algorithmic subtyping, which we will introduce later), so the rules are named (DS-*Name*). Our naming scheme for rules distinguishes between *congruence rules* like (DS-APP), which compare terms of similar shape (i.e. functions with functions, or applications with applications), and *reduction rules* like (DS-EAPP), which compare terms of different shape. Reduction rules are named (DS-E*Name*).

In order to make the presentation as compact as possible, we use $\triangleleft$ as a metavariable that ranges over both the subtype relation ($\leq$), and the type equivalence relation ($\equiv$). (This particular presentation style is the reason why we call the calculus System $\lambda_{\triangleleft}$.) Each rule that is given in terms of $\triangleleft$ thus defines two different rules. For example, the rule for application (DS-APP) actually denotes the following:

$$\frac{\Gamma \vdash\ t \triangleleft t', \quad u \equiv u'}{\Gamma \vdash\ t(u) \triangleleft t'(u')} \quad \text{means} \quad \frac{\Gamma \vdash\ t \leq t', \quad u \equiv u'}{\Gamma \vdash\ t(u) \leq t'(u')} \quad \text{and} \quad \frac{\Gamma \vdash\ t \equiv t', \quad u \equiv u'}{\Gamma \vdash\ t(u) \equiv t'(u')}$$

Subtyping between functions is pointwise. If f and g are functions, then $f \leq g$ if and only if $f(a) \leq g(a)$ for every valid argument a. This notion of point-wise subtyping is exactly the same as in System $F^{\omega}_{\leq}$.

The argument type of a function is invariant rather than contravariant in subtypes. That is to say, $\lambda x \leq t.\ u \ \leq\ \lambda x \leq t'.\ u$ only if $t \equiv t'$. This approach corresponds to subtyping in kernel $F^{\omega}_{\leq}$. Although contravariance is a potentially useful extension, it also results in several well-known pathologies [Pierce, 1994]. None of the real-world examples that we discuss in later chapters require contravariance, so we do not include it in the theory. Contravariance is discussed in more detail in Section 2.9.8.

2.3.2 Well-formedness

The subtype relation is defined only over terms that are well-formed. For clarity of presentation, however, we have chosen to define the $\lhd$ relation syntactically, omitting the well-formedness checks. The $\lhd$ relation is thus not a complete definition of subtyping. The complete subtype relation is written as $t \leq_{\text{wf}} u$, (pronounced "t is a well-subtype of u"). As will be shown in the next section, any derivation of $t \leq_{\text{wf}} u$ will only compare well-formed subterms of t and u.

The well-formedness judgement makes the following type checks:

- It ensures that the program is well-scoped: every variable x must be defined within the typing context.
- In a function application $t(u)$, it ensures that t is a function, and that u is a subtype of the argument type of that function.

2.3.3 Example: $3 \leq \mathsf{Nat}$

To illustrate how System $\lambda_{\lhd}$ works, we will show that 3 is a subtype of Nat, using the standard encoding for natural numbers and Church numerals [Pierce, 2002]. Note that System $\lambda_{\lhd}$ does not have arrow-types; instead, the standard arrow-type $t \to u$ is syntax sugar for $\lambda y \leq t.\ u$, where y is chosen fresh, and does not appear within u.

$$
\begin{array}{lcll}
\mathsf{Nat} & = & \lambda x \leq \mathsf{Top}.\ (x \to x) \to x \to x & \text{(sugared)} \\
 & = & \lambda x \leq \mathsf{Top}.\ \lambda f \leq (\lambda y \leq x.\ x).\ \lambda a \leq x.\ x & \text{(de-sugared)} \\
3 & = & \lambda x \leq \mathrm{Top}.\ \lambda f \leq (\lambda y \leq x.\ x).\ \lambda a \leq x.\ f(f(f(a))) &
\end{array}
$$

In the above example, x is a type variable, while f and a are object variables. There is no syntactic difference between the two – the difference lies in how they are used. To show that $3 \leq \mathsf{Nat}$, we show that $\Gamma,\ f \leq (\lambda y \leq x.\ x),\ a \leq x \vdash\ f(f(f(a))) \leq x$, as follows:

$$
\begin{array}{l}
\Gamma,\ f \leq (\lambda y \leq x.\ x),\ a \leq x \vdash \\
\qquad
\begin{array}{lcll}
f(f(f(a))) & \leq & (\lambda y \leq x.\ x)(f(f(a))) & \text{by DS-App, DS-EVar} \\
 & \equiv & x & \text{by DS-EApp}
\end{array}
\end{array}
$$

Note that encoding shown above is not specific to System $\lambda_{\lhd}$; it would be perfectly valid in System $\mathrm{F}^{\omega}_{\leq}$ as well. The fact that 3 is a subtype of Nat is a simple consequence

of our decision to encode natural numbers at the level of types, rather than the level of objects.

2.3.4 Adding Universes

By showing that 3 is a subtype of Nat, we have shown that subtyping can replace typing to a certain degree. The subtype judgement tells us that 3 and Nat have a similar shape; they are both functions, and they both accept the same number and the same type of arguments as input. This definition is sufficient for us to build a static type system.

However, there is a practical reason why one might wish to distinguish between 3 and Nat, which has nothing to do with shape. When performing a computation, one might like to know that the result of a given compuation will be *concrete*. Given a function f of type $\mathsf{Nat} \to \mathsf{Nat}$, we want to know that that $f(3)$ will return an actual number, like 0 or 5, rather than a type, like Nat.

This distinction is easy to add to any of the type systems discussed in this thesis; a solution for System $\lambda_{\triangleleft}$ is shown here. We divide terms into two universes: 0 is the universe of objects, and 1 is the universe of types. To distinguish between these universes, the syntax of System $\lambda_{\triangleleft}$ must be extended so that variables are tagged with their universe. Object object variables are written as x^0 or y^0, while type variables are written as x^1 or y^1.

$$\begin{array}{ll} J,K & ::= 0 \mid 1 \\ s,t,u & ::= x^K \mid \mathsf{Top} \mid \lambda x^K \leq t.\ u \mid t(u) \end{array}$$

Once variables have been tagged with their universe, the judgement $t \in \mathcal{U}(K)$ determines whether a given term is an object or a type:

$$\begin{array}{lcll} x^K & \in & \mathcal{U}(K) & \\ \mathsf{Top} & \in & \mathcal{U}(1) & \\ \lambda x^J \leq t.\ u & \in & \mathcal{U}(K) & \text{if } u \in \mathcal{U}(K) \\ t(u) & \in & \mathcal{U}(K) & \text{if } t \in \mathcal{U}(K) \end{array}$$

Note that a function is in universe K if its body is in K, regardless of what universe its argument is in. This simple model allows any universe to predicate over any other, and thus supports both parametric polymorphism (i.e. objects that depend on types) and dependent types (i.e. types that depend on objects).

The well-formedness rule for function application must be modified to ensure that function arguments are in the correct universe:

$$\frac{\Gamma \vdash\ t \leq_{\text{wf}} (\lambda x^K \leq s.\ \mathsf{Top}), \quad u \leq_{\text{wf}} s \quad u \in \mathcal{U}(K)}{\Gamma \vdash\ t(u)\ \text{wf}}$$

It is easy to see that universes are preserved under β-reduction, because a variable in universe K is replaced with a term in universe K. Moreover, by extending our definitions of 3 and Nat with universe tags, it also clear that 3 is an object (i.e. $3 \in \mathcal{U}(0)$), and Nat is a type:

$$\begin{aligned} \mathsf{Nat} &= \lambda x^1 \leq \mathsf{Top}.\ \lambda f^0 \leq (x^1 \to x^1).\ \lambda a^0 \leq x^1.\ x^1 \\ 3 &= \lambda x^1 \leq \mathrm{Top}.\ \lambda f^0 \leq (x^1 \to x^1).\ \lambda a^0 \leq x^1.\ f^0(f^0(f^0(a^0))) \end{aligned}$$

With these definitions in place, we can rest assured that if $t \leq \mathsf{Nat}$, and $t \in \mathcal{U}(0)$, then the result of evaluating t will yield an actual number, rather than a type.

We present universes as a curiosity. The universe judgement shown here is completely orthogonal to subtyping, so the presence or absence of universes does not affect any of the results that we present in this thesis. The subtype relation is still defined over all terms in all universes. In particular, subtyping can cross universe boundaries: objects are still subtypes of types. Top is still a universal supertype that ranges over all terms, so System $\lambda_{\triangleleft}$ with universes remains fully impredicative.

For simplicity, we omit universes from System $\lambda_{\triangleleft}$, and from the DEEP-- and DEEP calculi that are presented in later chapters. Our purpose in describing them here is merely to illustrate that pure subtype systems are not limited to a single universe, and that it is still possible to distinguish between objects and types in practical programming if need be.

2.4 Embeddings of other languages into System $\lambda_{\triangleleft}$

The previous section provided a simple proof that $3 \leq \mathsf{Nat}$. This section explores the relationship between System $\lambda_{\triangleleft}$ and traditional type theory in more detail, by providing an embedding of two well-known type systems. We first demonstrate the way in which System $\lambda_{\triangleleft}$ combines typing and subtyping by embedding System $F_{\leq}$. We then demonstrate the way in which System $\lambda_{\triangleleft}$ handles dependent and higher-order types by embedding System $\lambda*$, the pure type system with $* : *$.

The embedding of System $\lambda*$ is particularly important, because this system is not strongly normalizing. As a result, System $\lambda_{\triangleleft}$ is not strongly normalizing either, a result that has important consequences for the metatheory presented later in this chapter.

It is important to note that these embeddings are not conservative. We will show that every term which is well-typed in System $F_{\leq}$ or System $\lambda*$ has a translation that is well-formed in System $\lambda_{\lhd}$. However, the converse is not true; there exist terms that are well-formed in System $\lambda_{\lhd}$, but are not well-typed in Systems $\lambda*$ or $F_{\leq}$.

One reason the embeddings are not conservative is simply because System $\lambda_{\lhd}$ is a more powerful theory; e.g. it includes subtyping, whereas System $\lambda*$ does not. A more important reason, however, is due to the nature of the embeddings. We translate every Π type to a λ abstraction. However, Π-types cannot be applied to arguments, whereas λ abstractions can be; it thus follows that System $\lambda_{\lhd}$ admits terms that would not ordinarily be allowed.

2.4.1 Embedding of System $F_{\leq}$

We have chosen to show an embedding of System $F_{\leq}$ rather than $F^{\omega}_{\leq}$ because $F_{\leq}$ is a much simpler system. The embedding itself is straightforward, and can be easily extended to System $F^{\omega}_{\leq}$. Our goal is to illustrate the way in which typing and subtyping can be combined, and we feel that the added complexity of working with $F^{\omega}_{\leq}$ would obscure rather than clarify the basic principles. Moreover, our translation of kinding and higher-order types can be seen in the embedding of System $\lambda*$, which is given in the next section.

The syntax of System $F_{\leq}$ is shown in Figure 2.3, along with the rules for typing and subtyping. Our presentation follows that of Pierce in [Pierce, 2002], with a few minor changes to syntax; for consistency, we write function application as $t(u)$, and use Π instead of $\forall$.

We have made one major change to system $F_{\leq}$: the arrow types in Figure 2.3 are invariant, rather than contravariant on the argument type. The definition of kernel $F_{\leq}$ given by Pierce uses invariant Π-types in order to preserve decidability, but arrow-types remain contravariant [Pierce, 1994]. However, System $\lambda_{\lhd}$ does not allow contravariance in any form, either on arrow-types or Π-types, since both arrow-types and Π-types are mapped to ordinary λ abstractions. System $\lambda_{\lhd}$ is thus strictly weaker than kernel $F_{\leq}$ in this respect.

To perform the embedding, we define a translation function $\langle\_\rangle$, which translates terms in System $F_{\leq}$ to System $\lambda_{\lhd}$. The translation function is shown in figure 2.4. Both objects and types in $F_{\leq}$ map to terms in $\lambda_{\lhd}$, and contexts Γ in $F_{\leq}$ map to contexts Γ in $\lambda_{\lhd}$. For simplicity, we assume that the set of variable names for System $\lambda_{\lhd}$ includes

x, y, z	Object variable	$T, U ::=$	Types
X, Y, Z	Type variable	X	type variable
		Top	universal supertype
$s, t, u ::=$	Object terms	$T \to U$	function type
x	variable	$\Pi X \leq T.\ U$	polymorphic type
$\lambda x : T.\ u$	function		
$\lambda X \leq T.\ u$	polymorphic function	$\Gamma ::=$	Contexts
$t(u)$	application	$\emptyset$	empty context
$t(T)$	type application	$\Gamma, x : T$	variable
		$\Gamma, X \leq T$	type variable

Typing: $\boxed{\Gamma \vdash_{\mathrm{F}} t : T}$

$$\frac{x : T \in \Gamma}{\Gamma \vdash_{\mathrm{F}} x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, x : T \vdash_{\mathrm{F}} u : U}{\Gamma \vdash_{\mathrm{F}} \lambda x : T.\ u\ :\ T \to U} \quad \text{(T-FUN)}$$

$$\frac{\Gamma, X \leq T \vdash_{\mathrm{F}} u : U}{\Gamma \vdash_{\mathrm{F}} \lambda X \leq T.\ u\ :\ \Pi X \leq T.\ U} \quad \text{(T-TFUN)}$$

$$\frac{\Gamma \vdash_{\mathrm{F}} t : U \to S, \quad u : U}{\Gamma \vdash_{\mathrm{F}} t(u) : S} \quad \text{(T-APP)}$$

$$\frac{\Gamma \vdash_{\mathrm{F}} t : \Pi X \leq U.\ S, \quad T \leq U}{\Gamma \vdash_{\mathrm{F}} t(T) : [X \mapsto T]U} \quad \text{(T-TAPP)}$$

$$\frac{\Gamma \vdash_{\mathrm{F}} t : T, \quad T \leq U}{\Gamma \vdash_{\mathrm{F}} t : U} \quad \text{(T-SUBS)}$$

Subtyping: $\boxed{\Gamma \vdash_{\mathrm{F}} T \leq U}$

$$\Gamma \vdash_{\mathrm{F}} T \leq T \quad \text{(S-REFL)}$$

$$\frac{\Gamma \vdash_{\mathrm{F}} S \leq T, \quad T \leq U}{\Gamma \vdash_{\mathrm{F}} S \leq U} \quad \text{(S-TRANS)}$$

$$\frac{X \leq T \in \Gamma}{\Gamma \vdash_{\mathrm{F}} X \leq T} \quad \text{(S-VAR)}$$

$$\Gamma \vdash_{\mathrm{F}} T \leq \mathsf{Top} \quad \text{(S-TOP)}$$

$$\frac{\Gamma \vdash_{\mathrm{F}} U \leq S}{\Gamma \vdash_{\mathrm{F}} T \to U \leq T \to S} \quad \text{(S-ARR)}$$

$$\frac{\Gamma, X \leq T \vdash_{\mathrm{F}} U \leq S}{\Gamma \vdash_{\mathrm{F}} \Pi X \leq T.\ U\ \leq\ \Pi X \leq T.\ S} \quad \text{(S-PI)}$$

Figure 2.3: System $\mathrm{F}_{\leq}$, without contravariant arrow-types.

both the names of object variables x, and type variables X in System $\mathrm{F}_{\leq}$. By making this assumption, we avoid the complications associated with renaming.

We now show that typing and subtyping are preserved under this translation. One minor complication is the fact that System F_{ω} as defined here makes no guarantees about the scoping of type variables; the language of types is sufficiently simple that none are needed. As a result, our translation only preserves typing and subtyping for well-scoped terms and contexts.

Definition 2.4.1 (Well-scoped contexts) We say that a context Γ in System $\mathrm{F}_{\leq}$ is well-scoped according to the following rules, where $\mathrm{fv}(T)$ denotes the set of free vari-

$$
\begin{array}{rcll}
\langle x \rangle & = & x & \\
\langle \lambda x : T.\ u \rangle & = & \lambda x \leq \langle T \rangle.\ \langle u \rangle & \\
\langle \lambda X \leq T.\ u \rangle & = & \lambda X \leq \langle T \rangle.\ \langle u \rangle & \\
\langle t(u) \rangle & = & \langle t \rangle(\langle u \rangle) & \\
\langle t(U) \rangle & = & \langle t \rangle(\langle U \rangle) & \\
 & & & \\
\langle X \rangle & = & X & \\
\langle \mathsf{Top} \rangle & = & \mathsf{Top} & \\
\langle T \to U \rangle & = & \lambda y \leq \langle T \rangle.\ \langle U \rangle & \text{where } y \text{ chosen fresh.} \\
\langle \Pi X \leq T.\ U \rangle & = & \lambda X \leq \langle T \rangle.\ \langle U \rangle & \\
 & & & \\
\langle \emptyset \rangle & = & \emptyset & \\
\langle \Gamma, x : T \rangle & = & \langle \Gamma \rangle, x \leq \langle T \rangle & \\
\langle \Gamma, x \leq T \rangle & = & \langle \Gamma \rangle, X \leq \langle T \rangle &
\end{array}
$$

Figure 2.4: Translation from System $F_{\leq}$ to System $\lambda_{\lhd}$

---

ables in T, and $\mathrm{dom}(\Gamma)$ denotes the set of variables defined in Γ:

$\emptyset$	is well-scoped
$\Gamma, X : T$	is well-scoped if $\mathrm{fv}(T) \subseteq \mathrm{dom}(\Gamma)$ and $X \notin \mathrm{dom}(\Gamma)$
$\Gamma, X \leq T$	is well-scoped if $\mathrm{fv}(T) \subseteq \mathrm{dom}(\Gamma)$ and $X \notin \mathrm{dom}(\Gamma)$

Lemma 2.4.2 (Substitution is preserved under translation)

$\langle [X \mapsto T]U \rangle = [X \mapsto \langle T \rangle]\langle U \rangle$

$\langle [x \mapsto t]u \rangle = [x \mapsto \langle t \rangle]\langle u \rangle$

Proof: By straightforward induction on U and u, respectively. □

Lemma 2.4.3 (Types in System $F_{\leq}$ are well-formed in System $\lambda_{\lhd}$)

For all T, If Γ wf and $\mathrm{fv}(T) \subseteq \mathrm{dom}(\Gamma)$, then $\Gamma \vdash \langle T \rangle$ wf.

Proof: By induction on the size of T.

Case $T = \mathsf{Top}$	by W-TOP.
Case $T = X$	by W-VAR; we know that $X \in \Gamma$ because $\mathrm{fv}(T) \subseteq \mathrm{dom}(\Gamma)$.
Case $T = U \to S$	by W-FUN; we have U wf and S wf by induction.
Case $T = \Pi X \leq T.\ U$	by W-FUN, similarly.

□

Lemma 2.4.4 (Contexts in System $F_{\leq}$ are well-formed in System $\lambda_{\lhd}$)

If Γ is a well-scoped context in System $F_{\leq}$, then $\langle \Gamma \rangle$ wf in System $\lambda_{\lhd}$.

Proof: By induction on the size of Γ, and lemma 2.4.3. □

Theorem 2.4.5 (Typing and subtyping in System $F_{\leq}$ are preserved under translation.)

If $\Gamma \vdash_F t : T$, Γ is well-scoped, and $\mathrm{fv}(t) \in \mathrm{dom}(\Gamma)$, then $\langle\Gamma\rangle \vdash \langle t\rangle \leq_{\mathrm{wf}} \langle T\rangle$.
If $\Gamma \vdash_F T \leq U$, Γ is well-scoped, $\mathrm{fv}(T) \subseteq \mathrm{dom}(\Gamma)$, and $\mathrm{fv}(U) \subseteq \mathrm{dom}(\Gamma)$,
then $\langle\Gamma\rangle \vdash \langle T\rangle \leq_{\mathrm{wf}} \langle U\rangle$.

Proof: By induction on the derivation of $t : T$ and $T \leq U$. The well-formedness of types and contexts is ensured by lemmas 2.4.3 and 2.4.4. Thus we need only prove that $\langle t\rangle$ wf, $\langle t\rangle \leq \langle T\rangle$, and $\langle T\rangle \leq \langle U\rangle$.

Case (T-VAR) and (S-VAR): $x \leq_{\mathrm{wf}} \langle T\rangle$ by (DS-EVAR) and (W-VAR).

Case (T-FUN), (T-TFUN), (S-ARR), and (S-PI): All four cases are similar and follow by induction, using (DS-FUN) and (W-FUN).

Case (T-APP), (T-TAPP): Both cases are similar; we describe (T-TAPP).
We have $t = t_1(S_1)$ and $T = [X \mapsto S_1]T_2$, where $t_1 : \Pi X \leq S_2.\ T_2$ and $S_1 \leq S_2$.
By the induction hypothesis, we derive: $\langle t_1\rangle \leq_{\mathrm{wf}} (\lambda X \leq \langle S_2\rangle.\ \langle T_2\rangle)$ and $\langle S_1\rangle \leq_{\mathrm{wf}} \langle S_2\rangle$.
We have $\langle t\rangle$ wf by (W-APP). Using rules (DS-APP), (DS-TRANS), and (DS-EAPP), we can derive that: $\langle t_1(S_1)\rangle \leq (\lambda X \leq \langle S_2\rangle.\ \langle T_2\rangle)(\langle S_1\rangle) \equiv [X \mapsto \langle S_1\rangle]\langle T_2\rangle$
The result follows from lemma 2.4.2 (substitution is preserved under translation).

Case (T-SUBS), (S-TRANS): by induction, using (DS-TRANS).

Case (S-REFL): $\langle T\rangle \leq_{\mathrm{wf}} \langle T\rangle$ by induction on the structure of T, using (DS-VAR) and (DS-TOP) for the base cases, and (DS-APP) and (DS-FUN) for the inductive cases.

Case (S-TOP): by (DS-ETOP). □

2.4.2 Symmetry in theories: an analogy with physics

The embedding of System $F_{\leq}$ exploits a symmetry between the rules for typing, and the rules for subtyping. In Figure 2.3, notice that the typing rule for polymorphic functions (T-TFUN) is very similar to the subtyping rule for the types of those functions (S-PI). If we merely replace Π with λ, and : with $\leq$, then the two rules become identical. Rule (T-FUN) and (S-ARR) are likewise similar, although less obviously so; the use

of arrow-types rather than Π-types tends to obscure the pattern. In System $F^{\omega}_{\leq}$ (not shown here), the subtype rule for higher-order types (which compares types of the form $\lambda X \leq T.\ U$) is yet a third variation on the same pattern.

Moreover, these similarities are not confined to functions and function types. The subsumption rule (T-SUBS), and the transitivity rule (S-TRANS) have a similar form, as do the rules for variables (T-VAR) and (S-VAR).

In general, there seems to be a great deal of symmetry between the typing rules and the subtyping rules of System $F^{\omega}_{\leq}$. Out of the six typing rules, four can be paired up with subtyping rules that are almost identical. The only two typing rules that do not have obvious counterparts are the ones for application: (T-APP) and (T-TAPP). These can be seen as a combination of two separate subtyping rules:

$$\frac{\Gamma \vdash_{\mathrm{F}} t : \Pi X \leq U.\ S, \quad T \leq U}{\Gamma \vdash_{\mathrm{F}} t(T) : [X \mapsto T]U} \text{ maps to } \frac{\Gamma \vdash t \leq t'}{\Gamma \vdash t(u) \leq t'(u)} \text{ and } \Gamma \vdash (\lambda x \leq t.\ u)(s) \equiv [x \mapsto s]u$$

In some other disciplines, most notably physics, the presence of unexplained symmetries within a theory is often interpreted as an indication that there is a deeper, underlying theory. For example, the so-called "Standard Model" of physics describes all of the fundamental forces and particles. For some strange reason, all of the particles come in sets of three, where the three members of each set differ only in mass. The Standard Model is the most accurate physical theory ever constructed, and has been verified by countless experimental results. Nevertheless, the unexplained symmetry within the theory is widely regarded by many physicists as strong evidence that the Standard Model is incomplete, and an enormous amount of time and effort has been devoted to finding a replacement [Weinberg, 1994].

It may seem somewhat far-fetched to draw an analogy between the Standard Model of physics and System $F_{\leq}$, but we do so because there is a lesson to be learned. The Standard Model has not yet been overturned, even after decades of intense and sustained effort by many of the brightest minds in the world. There is no shortage of alternative theories, from "string theory" and "supersymmetry" to "loop quantum gravity" [Smolin, 2002], but all such theories have thus far been either inconsistent with observation or mathematically intractable. The lesson from physics is that it is easy to observe that a symmetry exists, but it may be extremely difficult to construct an alternative theory that exploits that symmetry.

Our experience with System $\lambda_{\lhd}$ would seem to indicate that this principle extends

x, y, z	Variables		
$s, t, u ::=$	Terms	$\Gamma ::=$	Contexts
$*$	single universe	$\emptyset$	empty context
x	variable	$\Gamma, x : t$	variable
$\lambda x : t.\ u$	function		
$\Pi x : t.\ u$	pi-type		
$t(u)$	application		

Typing: $\boxed{\Gamma \vdash_P t : T}$

$$\emptyset \vdash_P * : * \quad \text{(T-Axiom)}$$

$$\frac{\Gamma \vdash_P t : * \quad x \notin \mathrm{fv}(\Gamma)}{\Gamma, x : t \vdash_P x : t} \quad \text{(T-Var)}$$

$$\frac{\Gamma \vdash_P u : s, \quad t : * \quad x \notin \mathrm{fv}(\Gamma)}{\Gamma, x : t \vdash_P u : s} \quad \text{(T-Weak)}$$

$$\frac{\Gamma \vdash_P t : * \quad \Gamma, x : t \vdash_P u : *}{\Gamma \vdash_P \Pi x : t.\ u\ :\ *} \quad \text{(T-Pi)}$$

$$\frac{\Gamma, x : t \vdash_P u : s \quad \Gamma \vdash_P (\Pi x : t.\ s) : *}{\Gamma \vdash_P (\lambda x : t.\ u) : (\Pi x : t.\ s)} \quad \text{(T-Fun)}$$

$$\frac{\Gamma \vdash_P t : \Pi x : u'.\ s, \quad u : u'}{\Gamma \vdash_P t(u) : [x \mapsto u]s} \quad \text{(T-App)}$$

$$\frac{\Gamma \vdash_P t : u, \quad s : * \quad u =_\beta s}{\Gamma \vdash_P t : s} \quad \text{(T-Beta)}$$

Figure 2.5: System $\lambda*$, the pure type system with $* : *$.

to computer science as well (although we do *not* pretend that the problem we face is anywhere near as difficult as replacing The Standard Model). As we shall show in the next section, System $\lambda_\triangleleft$ is not strongly normalizing. In following sections, we shall further show that the metatheory for System $\lambda_\triangleleft$ has thus far proved to be intractable. The work presented here can therefore best be seen as the first step on what may be a long road towards a fully symmetric theory.

2.4.3 Embedding of $\lambda*$, the pure type system with $* : *$

We now show that we can embed System $\lambda*$, the pure type system with $* : *$, into System $\lambda_\triangleleft$. The syntax and type rules for System $\lambda*$ are drawn from [Barendregt, 1992]. As before, we define a translation function $\langle \_ \rangle$, and show that the translation preserves typing (System $\lambda*$ does not have a subtype relation). The translation is shown in Figure 2.6.

$$
\begin{array}{rcl}
\langle x \rangle & = & x \\
\langle * \rangle & = & \mathsf{Top} \\
\langle \lambda x : t.\ u \rangle & = & \lambda x \leq \langle t \rangle.\ \langle u \rangle \\
\langle \Pi x : t.\ u \rangle & = & \lambda x \leq \langle t \rangle.\ \langle u \rangle \\
\langle t(u) \rangle & = & \langle t \rangle (\langle u \rangle) \\
 & & \\
\langle \emptyset \rangle & = & \emptyset \\
\langle \Gamma, x : T \rangle & = & \langle \Gamma \rangle, x \leq \langle T \rangle
\end{array}
$$

Figure 2.6: Translation from System $\lambda*$ to System $\lambda_{\triangleleft}$

Lemma 2.4.6 (Substitution is preserved under translation)
$\langle [x \mapsto t]u \rangle = [x \mapsto \langle t \rangle]\langle u \rangle$

Proof: By straightforward induction on u. □

Theorem 2.4.7 (Typing in System $\lambda*$ is preserved under translation)
If $\Gamma \vdash_{\mathrm{P}} t : u$ then $\langle \Gamma \rangle \vdash \langle t \rangle \leq_{\mathrm{wf}} \langle u \rangle$.

Proof: By induction on the derivation of $t : u$.

The type formation rules of System $\lambda*$ ensure that every derivation of the form $\Gamma_1, x : s, \Gamma_2 \vdash_{\mathrm{P}} t : u$ contains $\Gamma_1 \vdash_{\mathrm{P}} s : *$ as a subderivation. By the induction hypothesis, we thus know that the type context is well-formed, i.e. $\Gamma \vdash_{\mathrm{P}} t : u$ implies $\langle \Gamma \rangle$ wf. The rest of the proof proceeds by cases.

Case (T-AXIOM): $\mathsf{Top} \leq_{\mathrm{wf}} \mathsf{Top}$ by (DS-ETOP) and (W-TOP).

Case (T-VAR): $x \leq_{\mathrm{wf}} \langle u \rangle$ by (DS-EVAR) and (W-VAR).

Case (T-WEAK): by lemma 2.5.4 (weakening).

Case (T-PI): $(\lambda x \leq \langle t_1 \rangle.\ \langle t_2 \rangle) \leq_{\mathrm{wf}} \mathsf{Top}$ by induction, using (DS-ETOP) and (W-FUN).

Case (T-FUN): $(\lambda x \leq \langle t_1 \rangle.\ \langle t_2 \rangle) \leq_{\mathrm{wf}} (\lambda x \leq \langle t_1 \rangle.\ \langle t_2' \rangle)$ by induction, using (DS-FUN) and (W-FUN).

Case (T-APP): $t = t_1(s_1)$, $u = [x \mapsto s_1]t_2$, where $t_1 : \Pi x : s_2.\ t_2$ and $s_1 : s_2$.
By the induction hypothesis, we derive: $\langle t_1 \rangle \leq_{\mathrm{wf}} (\lambda x \leq \langle s_2 \rangle.\ \langle t_2 \rangle)$ and $\langle s_1 \rangle \leq_{\mathrm{wf}} \langle s_2 \rangle$.
We have thus have $\langle t \rangle$ wf by (W-APP). Using rules (DS-APP), (DS-TRANS), and (DS-EAPP), we can derive that: $\langle t_1(s_1) \rangle \leq (\lambda x \leq \langle s_2 \rangle.\ \langle t_2 \rangle)(\langle s_1 \rangle) \equiv [x \mapsto \langle s_1 \rangle]\langle t_2 \rangle$

The result then follows from lemma 2.4.6.

Case (T-BETA): If $u =_\beta s$, then $u \equiv s$ by zero or more applications of (DS-EAPP), along with (DS-TRANS) and (DS-SYM). The result then follows from (DS-TRANS).

□

2.4.4 Logical consistency and Girard's paradox

System $\lambda*$ suffers from a logical paradox known as *Girard's paradox* [Girard, 1972] [Barendregt, 1992]. According to the Curry-Howard isomorphism, every type system can be interpreted as a logic. Types correspond to logical propositions, while objects correspond to proofs of those propositions [de Groote (editor), 1995].

A type is said to be *inhabited* if an object of that type exists. Every inhabited type corresponds to a proposition that is true, since a proof that that proposition (i.e. the object) exists. When viewed as a logic through the lense of the Curry-Howard isomorphism, System $\lambda*$ is logically inconsistent, because every type is inhabited. In other words, every proposition is provable, including ones that are known to be false.

Since System $\lambda_\triangleleft$ eliminates the distinction between types and objects, it is unclear how the Curry-Howard isorphism could be meaningfully applied. Every "type" in System $\lambda_\triangleleft$ is trivially "inhabited" because it has itself as a subtype, but definining "inhabited" in this way is clearly nonsense. The "propositions as types" principle is a characteristic of the typing relation, not the subtype relation. In the absence of typing, we need some other definition of logical consistency.

Nevertheless, Girard's paradox has a very important computational consequence, regardless of how "logical consistency" is defined. System $\lambda*$ is not *strongly normalizing*; there are reduction sequences that do not terminate. Since System $\lambda*$ can be embedded in System $\lambda_\triangleleft$, System $\lambda_\triangleleft$ is not strongly normalizing either.

The lack of strong normalization is not necessarily cause for concern, because we intend to use the calculus for practical programing, not theorem-proving. In the next chapter we will add fixpoints and recursion to the calculus, which destroys both normalization and consistency far more thoroughly than Girard's paradox does.

However, the lack of strong normalization has certain consequences as far as the meta-theory is concerned. In particular, existing proofs of transitivity elimination have been done only for type systems that are strongly normalizing. Our attempts to extend these proof techniques to the non-strongly normalizing case have not succeeded.

2.4.5 Impredicativity

The fact that System $\lambda_{\triangleleft}$ suffers from Girard's paradox is not caused by the conflation of types and objects, as one might intuitively expect. System $\lambda*$ suffers from the same paradox, and System $\lambda*$ does distinguish between types and objects. One might argue that the circular $* : *$ typing rule is the source of the paradox because it confuses the universe of types and the universe of objects. However, as Barendregt explains in [Barendregt, 1992], Girard's paradox can also be found in System λU, another pure type system which has no such circularity.

The cause of Girard's paradox is impredicativity. A logical proposition is *impredicative* if it quantifies over itself. For example, let Q be the proposition "for all propositions P, P implies P". Since Q is itself a proposition, we can use Q to derive that Q implies Q. In a programming language, impredicativity is dangerous because it can be used to construct a function that can be applied to itself, and self-application is the basis for fixpoint combinators and other non-terminating expressions. Impredicativity must thus be handled very carefully if a type theory is to remain strongly normalizing.

It is possible for a type theory to be both impredicative and consistent; Girard's System F [Girard et al., 1989] and Luo's extended calculus of constructions (ECC) [Luo, 1994] are two well-known examples. However, both of these systems place tight controls on impredicativity. System F controls impredicativity by using a relatively simple structure, which does not support support higher-order or dependent types. ECC is a much richer and more complex theory, but it maintains consistency by using a carefully managed hierarchy of universes, in which impredicativity is only allowed at the level of propositions, and not over arbitrary data types.

System $\lambda_{\triangleleft}$ suffers from Girard's paradox because it places no constraints on impredicativity. All terms belong to a single universe, and Top is both a member of the universe, and a type that ranges over all elements of the universe. Because of this structure, it is very easy to construct functions that allow self-application; the identity function is a simple example:

$$\mathsf{id} = \lambda x \leq \mathsf{Top}.\ x$$
$$\mathsf{id}(\mathrm{id}) \longrightarrow \mathsf{id}$$

Constructing a looping combinator is much more difficult. Simple looping combinators like the ω-term or the fixpoint combinator, which are well-known from the untyped λ-calculus, are not well-formed in System $\lambda_{\triangleleft}$. For example, one might attempt to write the ω term as:

$$\omega = (\lambda x \leq \mathsf{Top}.\ x(x))(\lambda x \leq \mathsf{Top}.\ x(x))$$

This term is not well-formed because the application $x(x)$ is invalid; x must be a subtype of a function rather than Top. In general, it is not possible to construct a finite λ abstraction which allows self-application in this way, because such a term would have to have itself as its argument type.

Although it is possible to construct a non-terminating term using Girard's paradox, it is not easy. As Howe shows for System $\lambda*$, a direct encoding yields a looping combinator that is 40 pages long without white space, and it was only possible to construct it with computer assistance [Howe, 1987]. It is therefore highly unlikely that the paradox would be encountered by accident in practical programming.

2.5 Type Safety

The next few sections will seek to establish the most important theoretical result for System $\lambda_{\lhd}$: type safety. We have been able to construct a partial proof of type safety that is good enough to rule out any of the obvious counter-examples. Unfortunately, our proof is incomplete. We will discuss in detail the reasons why a complete proof has been so hard to develop. In the absence of either a complete proof, or a counter-example, the type safety of DEEP remains an open question.

The interested reader may wish to refer back to Figure 2.2, which gives the definition of declarative subtyping.

2.5.1 Basic meta-theoretic properties

We first introduce establish some basic meta-theoretic properties of subtyping. In all of our proofs, we use the following notation:

- Γ, Γ' is the concatenation of the definitions in Γ and Γ'.
- $[x \mapsto t]\Gamma$ denotes the substitution of the term t for the variable x within Γ.
- We use a, b, c, d in addition to s, t, u as meta-variables for terms;
  v, w are meta-variables for values.

Lemma 2.5.1 (Context formation)

If $\Gamma \vdash t$ wf, then Γ wf.

Proof: By induction on the derivation of t wf. The base cases are (W-VAR) and (W-TOP), where the result follows immediately. All other cases follow by induction. □

Lemma 2.5.2 (Free variables)

If $\Gamma = x_1 \leq t_1, ..x_n \leq t_n$, and Γ wf, then every $x_i^{i\in\{1..n\}}$ is distinct.

Proof: By induction on Γ wf. □

Lemma 2.5.3 (System $\lambda_{\lhd}$ is well-defined.)

If $\Gamma \vdash t \lhd_{\text{wf}} u$, and $\Gamma' \vdash s_1 \lhd s_2$ is a subderivation of that fact, then $\Gamma' \vdash s_1$ wf, s_2 wf. In other words, a well-subtyping derivation will only compare well-formed terms.

Proof: By induction on the derivation of $t \leq_{\text{wf}} u$.

We first show that t wf implies that every subterm of t is well-formed, and similarly for u; the proof follows by induction on the derivation of t wf. As a result, the recursive uses of $\lhd$ in (DS-FUN) and (DS-APP) are made between well-formed terms. For (DS-EVAR), the result follows from lemma 2.5.1 (Context formation). The only comparison which could potentially introduce a non-well-formed term is (DS-TRANS), which has well-formedness as a premise. □

Lemma 2.5.4 (Weakening)

(1) If $\Gamma \vdash t$ wf, and $\Gamma, \Gamma' \vdash u$ wf, and $x \notin \text{dom}(\Gamma, \Gamma')$, then $\Gamma, x \leq t, \Gamma' \vdash u$ wf.

(2) If $\Gamma \vdash t$ wf, and $\Gamma, \Gamma' \vdash u \lhd s$, and $x \notin \text{dom}(\Gamma, \Gamma')$, then $\Gamma, x \leq t, \Gamma' \vdash u \lhd s$.

Proof: By induction on derivations. □

Lemma 2.5.5 (Narrowing)

If $\Gamma, x \leq t, \Gamma' \vdash u \lhd_{\text{wf}} s$ and $\Gamma \vdash t' \leq_{\text{wf}} t$ then $\Gamma, x \leq t', \Gamma' \vdash u \lhd_{\text{wf}} s$.

Proof: By induction on the derivation of $u \lhd_{\text{wf}} s$. Every judgement of the form $x \leq t$ can be replaced by a judgement of the form $x \leq t' \leq t$, using rule (DS-TRANS). □

Lemma 2.5.6 (Substitution)

If $\Gamma, x \leq t, \Gamma' \vdash u \lhd_{\text{wf}} s$ and $\Gamma \vdash t' \leq_{\text{wf}} t$ then $\Gamma, [x \mapsto t']\Gamma' \vdash [x \mapsto t']u \lhd_{\text{wf}} [x \mapsto t']s$.

Proof: By induction on the derivation of $u \lhd_{\text{wf}} s$. Every derivation of the form x wf is replaced with t' wf, and every derivation of the form $x \leq t$ is replaced with $t' \leq t$. □

Lemma 2.5.7 ($\equiv$ is reflexive) $\Gamma \vdash t \equiv t$ for any Γ and t.

Proof: By induction on t, using rules (DS-VAR), (DS-TOP), (DS-FUN), and (DS-APP). □

Lemma 2.5.8 (Reduction implies equivalence)
If $t \longrightarrow t'$, then $\Gamma \vdash t \equiv t'$.

Proof: By induction on the derivation of $t \longrightarrow t'$.
Case $(\lambda x \leq a.\ b)(c) \longrightarrow [x \mapsto c]b$: by rule (DS-EAPP).
Case $C[u] \longrightarrow C[u']$: By induction, using rules (DS-APP) and (DS-FUN). □

2.5.2 The easy part of type safety

Our proof of type safety is an adaptation of the standard technique of progress and preservation [Wright and Felleisen, 2004]. In a traditional type system, progress states that "well-typed terms don't get stuck". If $t : T$, then t must either be a value, or be reducible to another term. Preservation states that reducing a term will not change its type; if $t : T$ and $t \longrightarrow t'$, then $t' : T$. Together, these two results show that the reduction sequence for any well-typed term will either go on forever, or terminate to yield a well-typed value.

In System $\lambda_{\lhd}$ we prove a similar result for well-formedness. In the case of progress, we show that if $\emptyset \vdash t$ wf, then either t is a value, or there exists a t' such that $t \longrightarrow t'$. For preservation, we must show that both well-formedness and subtyping are preserved under reduction. If $t \leq_{\text{wf}} u$ and $t \longrightarrow t'$ then $t' \leq_{\text{wf}} u$.

The title of this section is "the easy part of type safety" because the full proof of type safety depends on a crucial property of subtyping called *transivity elimination*. Transitivity elimination is the 800-pound gorilla hiding in the metatheory; it is far and away the most difficult result to prove. For now, we will take transitivity elimination to be a conjecture; it is the subject of Section 2.6.

Conjecture 2.5.9 (Transitivity elimination)
If $\Gamma \vdash v \leq_{\text{wf}} w$, then there exists a proof of $\Gamma \vdash v \leq w$ that ends in either (DS-FUN) or (DS-ETOP).

Lemma 2.5.10 (Inversion of subtyping — declarative version)
If $\Gamma \vdash (\lambda x \leq t.\ u) \leq_{\text{wf}} (\lambda x \leq t'.\ u')$ then $\Gamma \vdash t \equiv t'$.

Proof: By conjecture 2.5.9 (transitivity elimination). □

Theorem 2.5.11 (Progress)

If $\emptyset \vdash t$ wf then either $t = v$ for some v (i.e. t is a value), or there exists a t' such that $t \longrightarrow t'$.

Proof: By induction on the derivation of t wf.

Case (W-TOP): Immediate, because $t = v$.
Case (W-VAR): Can't happen, because x is not well-formed in an empty context.
Case (W-FUN): Immediate, because $t = v$.
Case (W-APP):
Subcase $t = t_1(t_2)$, where $t_1 \neq v$. We have $t_1 \longrightarrow t_1'$ by induction.
Subcase $t = v(t_2)$, where $t_2 \neq v$. We have $t_2 \longrightarrow t_2'$ by induction.
Subcase $t = \mathsf{Top}(c)$. Can't happen, because t is not well-formed.
By conjecture 2.5.9, there is no function which is a supertype of Top.
Subcase $t = (\lambda x \leq a.\ b)(c)$. Immediate, because $t \longrightarrow [x \mapsto c]b$.
□

Theorem 2.5.12 (Preservation)

If $\Gamma \vdash t \leq_{\text{wf}} u$ and $t \longrightarrow t'$ then $\Gamma \vdash t' \leq_{\text{wf}} u$.

Proof: By induction on the derivation of t wf.

The proof has two parts. For the first part of the proof, we show that t wf and $t \longrightarrow t'$ implies t' wf; this part proceeds by cases:

Case (W-TOP): Can't happen; TOP has no reducts.
Case (W-VAR): Can't happen; x has no reducts.
Case (W-FUN):
Subcase $t = \lambda x \leq t_1.\ t_2$ and $t_2 \longrightarrow t_2'$. By induction.
Subcase $t = \lambda x \leq t_1.\ t_2$ and $t_1 \longrightarrow t_1'$. By the induction hypothesis and lemma 2.5.8 (reduction implies equivalence), we have $t_1 \equiv_{\text{wf}} t_1'$. The typing context for t_2 changes from $\Gamma, x \leq t_1$ to $\Gamma, x \leq t_1'$; but t_2 remains well-formed by lemma 2.5.5 (narrowing).
Case (W-APP):
Subcase $t = t_1(t_2)$ and $t_1 \longrightarrow t_1'$. By induction.
Subcase $t = t_1(t_2)$ and $t_2 \longrightarrow t_2'$. By induction.
Subcase $t = (\lambda x \leq a.\ b)(c) \longrightarrow [x \mapsto c]b$.

The two premises of t wf are $(\lambda x \leq a.\ b) \leq_{\text{wf}} (\lambda x \leq a'.\ b')$, and $c \leq_{\text{wf}} a'$. We have $a \equiv_{\text{wf}} a'$ by lemma 2.5.10 (inversion of subtyping), which gives us $c \leq_{\text{wf}} a$ by rule (DS-TRANS). We then have $[x \mapsto c]b$ wf by lemma 2.5.6 (substitution).

For the second part of the proof, we must show that if $t \leq u$, and $t \longrightarrow t'$, then $t' \leq u$. By lemma 2.5.8 (reduction implies equivalence) we have $t \equiv t'$. We thus have $t' \leq u$ using rule (DS-TRANS), along with the fact that t wf. □

2.5.3 The trouble with transitivity

The definition of subtyping given in Figure 2.2 is known as *declarative subtyping*. The declarative definition is easy to read, and it is also easy to prove certain lemmas, such as substitution and narrowing. However, the declarative definition is problematic because it includes a transitivity rule:

$$\frac{\Gamma \vdash\ s \leq t, \quad t \leq u, \quad t\ \text{wf}}{\Gamma \vdash\ s \leq u}\ \text{(DS-TRANS)}$$

Transitivity is troublesome for two reasons. The first problem is that declarative subtyping is not an algorithm, because it is not syntax-directed. The transitivity rule introduces an arbitrary term t in the premises which is completely absent from the conclusion, so any attempt to use transitivity in a syntax-directed manner would need to "guess" an appropriate value for t. There is therefore a practical need to find a different formulation of subtyping that can be implemented within a compiler.

The second, more serious problem is that the presence of a transitivity rule prevents us from completing the proof of type safety. The type safety proof shown above has the following step: given a well-formed redex $(\lambda x \leq a.\ b)(c)$, we must show that $[x \mapsto c]b$ is well-formed. This seems like a straightforward application of the substitution lemma, but the substitution lemma requires that $c \leq a$. We do not actually have a proof that $c \leq a$; instead, well-formedness tells us that:

(1) $(\lambda x \leq a.\ b) \leq (\lambda x \leq a'.\ \mathsf{Top})$ and
(2) $c \leq a'$

If the derivation of (1) ends in rule (DS-FUN), then we have $a \equiv a'$, and consequently $c \leq a$, so the substitution lemma applies. However, if the derivation if (1) ends in rule (DS-TRANS), then there is no immediate relationship between a and a'. If a and a' are unrelated, then the substitution lemma cannot be applied, and the proof of

type safety cannot be completed.

The standard technique for resolving this problem is to reformulate the subtype relation into an algorithmic form that does not include a transitivity rule, a process called *transitivity elimination* [Pierce, 2002] [Steffen and Pierce, 1994]. In essence, transitivity elimination is a proof that the subtype relation is sound. After all, if the subtyping rules allowed us to derive that $(\lambda x \leq a.\ b) \leq (\lambda x \leq a'.\ \mathsf{Top})$, where $a \not\equiv a'$, then there would clearly be an error in the theory. As a result, transitivity elimination is an unavoidable part of type safety.

2.5.4 Transitivity elimination

The proof of transitivity elimination has two parts:

- Reformulate the subtype judgement into a algorithmic system that does not include a transitivity rule.
- Show that transitivity is admissible in the algorithmic system.

The proof that transitivity is admissible involves demonstrating that any subtype derivation which makes use of transitivity can be transformed into one that does not. In other words, it states that the transitivity rule is still valid, even though it is not included in the formal inference rules of the system.

There are two reasons why we we still need transitivity. First, we would like to know that the algorithmic formulation of subtyping is equivalent to the declarative formulation. Second, transitivity is used in certain key lemmas for type safety, such as narrowing and substitution (see Section 2.5.1).

The proof of transitivity elimination is fairly straightforward for simple type theories like System $F_{\leq}$, but it is much more difficult for any theory that includes higher order subtyping, such as System $F^{\omega}_{\leq}$, or System $\lambda_{\lhd}$. The fundamental difficulty is that higher-order subtyping involves β-reduction on types (i.e. rule (DSE-APP)), and β-reduction changes both the size and shape of terms. The change in shape complicates the proof by dramatically increasing the number of cases to consider, and the change in size makes it difficult to devise an appropriate induction hypothesis.

Previous approaches in the literature all deal with β-reduction in a similar manner. A "basic" subtype relation is defined over a subset of terms which have a restricted syntax: they must either be in normal form [Steffen and Pierce, 1994] [Compagnoni, 1995] [Chen, 1999] or weak head normal form [Compagnoni and Goguen, 2003]. Terms which are not in normal form must be reduced to normal form before the

"basic" subtype relation can be applied. Because the "basic" relation is restricted to normal forms, transitivity can be eliminated using standard techniques. Unfortunately, this approach cannot be used for System $\lambda_{\lhd}$, because System $\lambda_{\lhd}$ is not strongly normalizing or even weakly normalizing; there is no guarantee that the terms in question have normal forms.

Zwanenburg's formulation of subtyping [Zwanenburg, 1999] is similar in that terms must be reduced until they match a restricted syntax in which no redex occurs on the spine of the term. However, Zwanenburg's syntax does not quite correspond to a normal form, so his approach can be used in theories that are not strongly normalizing. Unfortunately, Zwanenburg's appoach does not permit bounded quantification on λ-abstractions (e.g. $\lambda x \leq t.\ u$); he defines subtyping only over λ-abstractions with kinded quantification (e.g. $\lambda x : K.\ u$).

These systems, and their relationship to System $\lambda_{\lhd}$, will be discussed in more detail in Section 2.9, at the end of this chapter. For now, suffice it to say that none of the existing approaches to transitivity elimination apply to System $\lambda_{\lhd}$, so we must look elsewhere. The remainder of this chapter introduces a new approach to transitivity elimination that we feel is especially promising. Since the fundamental difficulty is dealing with β-reduction, we turn to the theory of abstract reduction systems for inspiration.

2.6 Algorithmic Subtyping

The problems caused by the transitivity rule are not restricted to subtyping. The same problems arise in any system which has a non-trivial notion of equality, including all higher-order type theories. Equality is a relation which is reflexive, symmetric, and transitive. Because the symmetry and transitivity rules are not syntax-directed, a practical algorithm for determining equality must eliminate them in some way.

The standard algorithmic technique for determining equality is to reformulate the equations of a system as reduction rules in an *abstract reduction system* or (ARS). Abstract reduction systems do not have a symmetry rule at all, so that complication is neatly eliminated. ARSs do have a transitivity rule, but transitivity is restricted to a form which is syntax-directed. For any single reduction, written $t \longrightarrow t'$, the subterms in t' are uniquely determined by t, so it is not necessary for an algorithm to "guess" intermediate terms. The reflexive and transitive closure of reduction, written $t_0 \longrightarrow\!\!\!\!\!\rightarrow t_n$, is not quite syntax directed because for any given t_i, more than one reduction rule may

apply. However, the system can be made syntax directed by choosing an appropriate reduction strategy.

In a reduction system, two terms are considered to be equal only if they can be reduced to a common term. In other words, $t = u$ if and only if there exists an s such that $t \longrightarrow\!\!\!\!\!\rightarrow s \longleftarrow\!\!\!\!\!\leftarrow u$. If the reduction system is *confluent*, then symmetry and transitivity are both admissible.

We adopt this technique for the algorithmic subtype relation in System $\lambda_{\lhd}$. To our knowledge, subtyping has never before been defined as an abstract reduction system, even though subtyping is conceptually similar to type equivalence, and β-reduction is universally used to define equivalence. Unlike equivalence, subtyping is an inequality rather than an equality, and that affects the way in which we define the relation.

The algorithmic formulation of subtyping for System $\lambda_{\lhd}$ is presented in figure 2.7. Most of the rules in the declarative system, including (DS-VAR), (DS-TOP), (DS-APP) and (DS-FUN), involve comparisons between terms which have the same shape. These rules become congruence rules in the reduction system. That leaves three remaining rules to consider: (DS-EAPP), (DS-EVAR), and (DS-ETOP):

$$\Gamma \vdash\ (\lambda x \leq t.\ u)(s) \equiv [x \mapsto s]u\ \ \text{(DS-EAPP)}$$

$$\frac{x \leq t \in \Gamma}{\Gamma \vdash\ x \leq t}\ \text{(DS-EVAR)} \qquad t \leq \text{TOP}\ \ \text{(DS-ETOP)}$$

The first rule states that terms are equivalent under β-reduction. The second rule, called *variable promotion*, states that variables can be promoted to their bounding types. The third rule states that every term is a subtype of Top.

We reformulate these three rules as reduction rules. An equivalence reduction, written $t \xrightarrow{\equiv} t'$, denotes a rewrite step which produces a term t' that is equivalent to t. β-reduction falls into this category. Subtype reduction, written $t \xrightarrow{\leq} t'$, denotes a rewrite step which produces a term t' that is a supertype of t. Variable promotion (DS-EVAR) and Top-promotion (DS-ETOP) fall into this category.

An equivalence reduction step $t \xrightarrow{\equiv} t'$ can be applied anywhere in a term. A subtype reduction step $t \xrightarrow{\leq} t'$ can only be applied in positive (i.e. covariant) positions within a term. Positive positions are limited to function bodies and the left-hand side of applications. We express this requirement by means of two evaluation contexts: $E_{\equiv}$ may have a hole in any position, whereas $E_{\leq}$ may only have a hole in positive positions. Neither evaluation context can step inside a λ-abstraction. Unlike ordinary reduction, both subtype and equivalence reduction must be done within a context Γ that

Prevalidity: $\boxed{\Gamma \text{ prevalid}}$

$$\emptyset \text{ prevalid} \qquad \text{(P-CTX1)}$$

$$\frac{\Gamma \text{ prevalid} \quad x \notin \text{dom}(\Gamma) \quad \text{fv}(t) \subseteq \text{dom}(\Gamma)}{\Gamma, x \leq t \text{ prevalid}} \qquad \text{(P-CTX2)}$$

Subtyping: $\boxed{\Gamma \vdash_A t \lhd u}$

$$\frac{\Gamma \text{ prevalid}}{\Gamma \vdash_A t \lhd t} \qquad \text{(AS-REFL)}$$

$$\frac{\Gamma \vdash_A t \xrightarrow{\lhd} t', \quad t' \lhd u}{\Gamma \vdash_A t \lhd u} \qquad \text{(AS-LEFT)}$$

$$\frac{\Gamma \vdash_A u \xrightarrow{\equiv} u', \quad t \lhd u'}{\Gamma \vdash_A t \lhd u} \qquad \text{(AS-RIGHT)}$$

Transitive Subtyping: $\boxed{\Gamma \vdash_A t \lhd^* u}$

$$\frac{\Gamma \vdash_A s \lhd u}{\Gamma \vdash_A s \lhd^* u} \qquad \text{(AST-SUB)}$$

$$\frac{\Gamma \vdash_A s \lhd^* t, \quad t \lhd^* u}{\Gamma \vdash_A s \lhd^* u} \qquad \text{(AST-TRANS)}$$

Subtype reduction: $\boxed{\Gamma \vdash_A t \xrightarrow{\leq} t'}$

$$\frac{\Gamma \text{ prevalid} \quad x \leq t \in \Gamma}{\Gamma \vdash_A x \xrightarrow{\leq} t} \qquad \text{(SRS-PROM)}$$

$$\frac{\Gamma \text{ prevalid}}{\Gamma \vdash_A t \xrightarrow{\leq} \mathsf{Top}} \qquad \text{(SRS-TOP)}$$

Equivalence reduction: $\boxed{\Gamma \vdash_A t \xrightarrow{\equiv} t'}$

$$\frac{\Gamma \vdash_A s \leq^* t}{\Gamma \vdash_A (\lambda x \leq t.\, u)(s) \xrightarrow{\equiv} [x \mapsto s]u} \qquad \text{(SRE-APP)}$$

$$\frac{\Gamma \text{ prevalid}}{\Gamma \vdash_A \mathsf{Top}(t) \xrightarrow{\equiv} \mathsf{Top}} \qquad \text{(SRE-TOPAPP)}$$

Congruence rules: $\boxed{\Gamma \vdash_A t \xrightarrow{\lhd} t'}$

$$E_{\equiv} ::= [] \mid E_{\equiv}(t) \mid t(E_{\equiv}) \mid \lambda x \leq E_{\equiv}.\, t$$

$$E_{\leq} ::= [] \mid E_{\leq}(t)$$

$$\frac{\Gamma \vdash_A t \xrightarrow{\lhd} t'}{\Gamma \vdash_A E_{\lhd}[t] \xrightarrow{\lhd} E_{\lhd}[t']} \qquad \text{(SR-CONG)}$$

$$\frac{\Gamma, x \leq t \vdash_A u \xrightarrow{\lhd} u'}{\Gamma \vdash_A \lambda x \leq t.\, u \xrightarrow{\lhd} \lambda x \leq t.\, u'} \qquad \text{(SR-FUN)}$$

$$\frac{\Gamma \vdash_A t \xrightarrow{\equiv} t'}{\Gamma \vdash_A t \xrightarrow{\leq} t'} \qquad \text{(SR-EQ)}$$

$\lhd$ is a meta-variable which ranges over $\{\leq, \equiv\}$.

Figure 2.7: System $\lambda_{\lhd}$— algorithmic subtyping

.

assigns bounding types to variables. Rule (SR-FUN) is used to reduce terms inside λ abstractions.

Within this framework, we define type equivalence and subtyping as follows. We use $\xrightarrow{\lhd}\!\!\!\!\rightarrow$ to denote the reflexive and transitive closure of $\xrightarrow{\lhd}$, and $u \xleftarrow{\equiv} t$ means the same thing as $t \xrightarrow{\equiv} u$.

- $\Gamma \vdash_A t \equiv u$ iff $\Gamma \vdash_A t \overset{\equiv}{\longrightarrow\!\!\!\!\!\rightarrow} s \overset{\equiv}{\longleftarrow\!\!\!\!\!\leftarrow} u$ for some s.
- $\Gamma \vdash_A t \leq u$ iff $\Gamma \vdash_A t \overset{\leq}{\longrightarrow\!\!\!\!\!\rightarrow} s \overset{\equiv}{\longleftarrow\!\!\!\!\!\leftarrow} u$ for some s.

The definition of type equivalence is standard. Two terms t and u are equivalent if they both reduce to a common term. The definition of subtyping is similar. The only difference is that for subtyping, the reduction sequence for t may promote variables to supertypes, or subterms to Top as necessary.

Within this framework, basic meta-theoretic properties of subtyping follow directly from standard properties of the corresponding reductions. In particular, if $\overset{\equiv}{\longrightarrow}$ and $\overset{\leq}{\longrightarrow}$ commute, then subtyping is transitive. If $\overset{\equiv}{\longrightarrow}$ and $\overset{\leq}{\longrightarrow}$ are strongly normalizing, then the subtype judgement is decidable.

The alert reader may have noticed the presence of two unusual rules in Figure 2.7. First, there is a new reduction (SRE-TopApp), which is not present in the declarative system. Algorithmic subtyping is defined over all terms, not just well-formed terms, which means that we must supply an interpretation for ill-formed applications. This issue is discussed in Section 2.6.2.

Second, we have actually defined two different subtype relations. The $\lhd$ relation is algorithmic subtyping, which has no transitivity rule. The $\lhd^*$ relation is the transtive closure of $\lhd$. Our ultimate goal in the meta-theory will be to demonstrate that these two relations are equivalent; that transitivity is admissible for $\lhd$. Initially, however, we must define $\lhd$ and $\lhd^*$ as separate relations, for a subtle reason that is explained in Section 2.6.3. In particular, notice that rule (SRE-App) makes use of $\lhd^*$ in its premise.

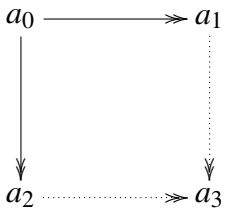
2.6.1 Transitivity and Confluence

Confluence is one of the most fundamental properties of any reduction system. A simple abstract reduction system $(A, \longrightarrow)$ consists of a set A of terms, and a reduction relation $\longrightarrow$ that relates elements of A. Confluence for a simple reduction system is defined as follows [Bezem et al., 2003]:

Definition 2.6.1 (Confluence)

Let $(A, \longrightarrow)$ be a simple abstract reduction system and let $\longrightarrow\!\!\!\!\!\rightarrow$ be the reflexive and transitive closure of $\longrightarrow$. This system is *confluent* if and only if the following holds. For every $a_0, a_1, a_2 \in A$, if $a_0 \longrightarrow\!\!\!\!\!\rightarrow a_1$ and $a_0 \longrightarrow\!\!\!\!\!\rightarrow a_2$, then there exists a $a_3 \in A$ such that $a_1 \longrightarrow\!\!\!\!\!\rightarrow a_3$ and $a_2 \longrightarrow\!\!\!\!\!\rightarrow a_3$.

Confluence is illustrated by the following diagram. Solid lines denote premises, and are called the *spanning edges* of the diagram. Dotted lines denote conclusions, and are called the *completing edges* of the diagram.



Confluence is an important property because it guarantees that normal forms are unique [Bezem et al., 2003]. A normal form is any term $a \in A$ that has no reducts; i.e. there is no a' such that $a \longrightarrow a'$. If a reduction system is confluent, then the order in which reductions are performed does not matter; all reduction paths will ultimately yield the same result.

2.6.1.1 Commutativity

The reduction system shown in Figure 2.7 is not a simple one, because we define two kinds of reduction: $\xrightarrow{\leq}$, and $\xrightarrow{\equiv}$. Moreover, the two kinds of reduction are not orthogonal; $\xrightarrow{\equiv}$ implies $\xrightarrow{\leq}$ by (SR-EQ). The exact property that we require for type safety is thus more specific than simple confluence.

The $\xrightarrow{\leq}$ relation by itself is trivially confluent, because (SRS-TOP) ($t \xrightarrow{\leq} \mathsf{Top}$) can be used as the completing edge of any diagram. The two non-trivial properties that we are interested in are shown below.

Conjecture 2.6.2 ($\xrightarrow{\equiv}$ is confluent)

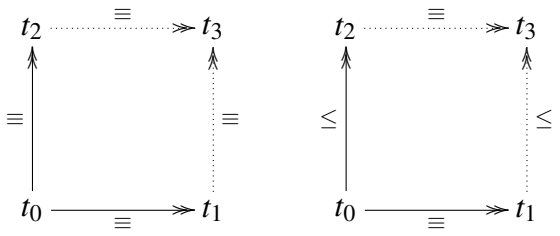
If $\Gamma \vdash_A t_0 \xrightarrow{\equiv} t_1$, $t_0 \xrightarrow{\equiv} t_2$, then there exists a t_3, such that $\Gamma \vdash_A t_1 \xrightarrow{\equiv} t_3$, $t_2 \xrightarrow{\equiv} t_3$.

Conjecture 2.6.3 ($\xrightarrow{\equiv}$ commutes with $\xrightarrow{\leq}$)

If $\Gamma \vdash_A t_0 \xrightarrow{\equiv} t_1$, $t_0 \xrightarrow{\leq} t_2$ then there exists a t_3, such that $\Gamma \vdash_A t_2 \xrightarrow{\equiv} t_3$, $t_1 \xrightarrow{\leq} t_3$.

It is not hard to prove that $\xrightarrow{\equiv}$ is confluent; the basic technique is the same as found in other calculi. However, it much harder to prove that $\xrightarrow{\equiv}$ and $\xrightarrow{\leq}$ commute. We are able to show that they commute locally, but we have thus far been unable to show that they commute globally; this issue will be discussed at length in Section 2.7.

The confluence and commutativity properties are illustrated by the following diagrams:



Throughout this thesis, we will write confluence diagrams such as these "upside-down" when compared to the way they are traditionally written. Because t_2 is a supertype of t_0, we feel that it is more intuitive to place it above t_0, so that the confluence diagram is oriented in the same way that an inheritance or subtype diagram would be.

Confluence and commutativity are similar properties, and both use the same proof techniques. The word "confluence" is used when only a single reduction relation is involved, while "commutativity" is used when two reduction relations are involved. In informal discussions, we will occassionally use the word "confluence" as a generic term that ranges over both properties, since the two are so similar.

2.6.1.2 Proof: commutativity implies transitivity

Transitivity elimination follows directly from the commutativity of $\xrightarrow{\leq}$ and $\xrightarrow{\equiv}$. We prove this fact as follows.

Lemma 2.6.4 (Inversion of subtyping: algorithmic system)

If $\Gamma \vdash_A \ t \lhd u$ then there exists an s such that $\Gamma \vdash_A \ t \xrightarrow{\lhd}\!\!\!\!\rightarrow s \xleftarrow{\equiv} u$.
Conversely, if $\Gamma \vdash_A \ t \xrightarrow{\lhd}\!\!\!\!\rightarrow s \xleftarrow{\equiv} u$ then $\Gamma \vdash_A \ t \lhd u$.

Proof: By induction on $t \lhd u$. □

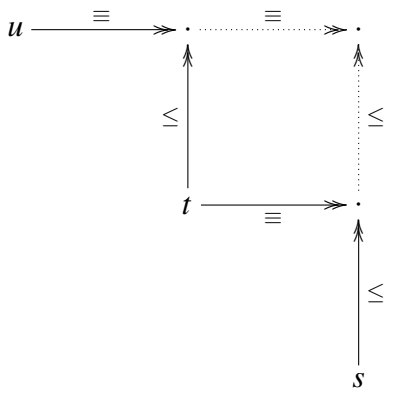
Corrollary: If $\Gamma \vdash_A (\lambda x \leq a.\ b) \leq (\lambda x \leq a'.\ b')$ then $\Gamma \vdash_A \ a \equiv a'$.

Proof: There exists a $(\lambda x \leq a''.\ b'')$, such that $a \xrightarrow{\equiv}\!\!\!\!\rightarrow a''$ and $a' \xrightarrow{\equiv}\!\!\!\!\rightarrow a''$. Thus, $\Gamma \vdash \ a \equiv a'$.
□

Lemma 2.6.5 (If $\xrightarrow{\equiv}$ commutes with $\xrightarrow{\leq}$ then transitivity is admissible)

Assume we have $\Gamma \vdash_A \ s \lhd t$ and $\Gamma \vdash_A \ t \lhd u$. If $\xrightarrow{\equiv}$ commutes with $\xrightarrow{\leq}$, then we can construct a derivation of $\Gamma \vdash_A \ s \lhd u$.

Proof: By lemma 2.6.4, according to the following diagram:



□

Lemma 2.6.6 (Transitivity elimination) If $\xrightarrow{\equiv}$ commutes with $\xrightarrow{\leq}$, then any derivation of $\Gamma \vdash_A t \lhd^* u$ can be transformed into a derivation of $\Gamma \vdash_A t \lhd u$.

Proof: By induction on the number of transitivity steps in the derivation of $t \lhd^* u$, using lemma 2.6.5. □

2.6.2 Subtyping as a syntactic relation

In the declarative system, subtyping is only defined over terms that are well-formed. This requirement sets up a circular dependency between subtyping and well-formedness that is very difficult to unravel. Circular dependencies of this kind arise frequently is type theories that support subtyping; Section 5.3.2 provides a summary of the literature. As is standard practice, we have attempted to unravel the dependency by defining an algorithmic version of subtyping that does not depend on well-formedness.

However, the algorithmic definition of subtyping introduces a new problem: ill-formed terms may now appear in subtype judgements. The type safety result for System $\lambda_{\lhd}$ states that $\xrightarrow{\equiv}$ preserves well-formedness. However, $\xrightarrow{\leq}$ does not preserve well-formedness, as is shown by the following counter-example:

$$(\lambda x \leq \mathsf{Top}.\ u)(s) \xrightarrow{\leq} \mathsf{Top}(s)$$

A particular reduction strategy for avoiding situations like the above is provided in Section 2.8. However, we cannot select a particular strategy when doing the confluence proofs, because confluence and commutativity, by definition, are supposed to show that all strategies will yield the same result. We thus accommodate ill-formed terms by making the following two changes:

$$\frac{\Gamma \vdash_A \ s \leq^* t}{\Gamma \vdash_A \ (\lambda x \leq t.\ u)(s) \xrightarrow{\equiv} [x \mapsto s]u} \qquad \Gamma \vdash_A \ \mathsf{Top}(t) \xrightarrow{\equiv} \mathsf{Top}$$

The first rule, (SRE-APP), places an additional premise on β-reduction, which prevents a function from being applied to an invalid argument. The second rule, (SRE-TOPAPP) allows Top to be used as a function. Although $\mathsf{Top}(t)$ is not well-formed, and this particular reduction would never appear in a well-subtyping judgement, we must be able to interpret applications of Top in order for the proof of local commutativity to go through (the relevant diagram is shown in Figure 2.10 (6)).

Rather than requiring that the context Γ be well-formed, the algorithmic definition of subtyping introduces a new judgement: Γ prevalid. Prevalidity ensures that variable names are unique and well-scoped, without requiring terms to be well-formed.

2.6.3 Conditional rewrite systems

Because β-reduction (rule SRE-APP) has an additional premise, the subtype relation that we have defined is an example of a *conditional rewrite system* [Bergstra and Klop, 1986]. A conditional rewrite system is one in which certain reduction rules have conditions attached to them. A classic example would be the following arithmetic rule, which might be used in an optimizing compiler:

$$t + u \longrightarrow 2 * t \text{ if } t = u$$

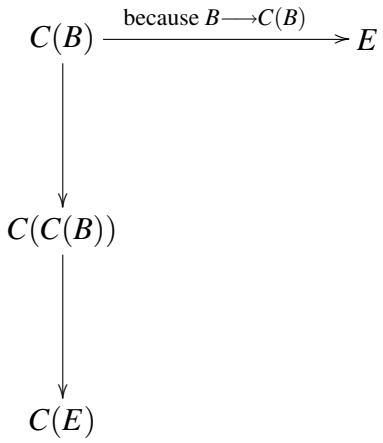
This is a circular definition. Reduction requires an equality check, but equality is defined in terms of reduction: $t = u$ iff $t \longrightarrow\!\!\!\!\!\rightarrow \cdot \longleftarrow\!\!\!\!\!\leftarrow u$. Such circularity can cause a failure of confluence in certain cases.

2.6.3.1 A possible counter-example to confluence

The classic counter-example, due to Bergstra and Klop [Bergstra and Klop, 1986], is as follows. Consider a conditional rewrite system described by the following rules:

$$\begin{array}{lcll} C(x) & \longrightarrow & E & \text{if } C(x) \longrightarrow\!\!\!\!\!\rightarrow \cdot \longleftarrow\!\!\!\!\!\leftarrow x \\ B & \longrightarrow & C(B) & \end{array}$$

If the condition on C were removed, then the above rewrite system would be confluent, as can be trivially demonstrated. However, in the presence of the condition, these rules give rise to the following confluence diagram, which cannot be completed:



If the counter-example shown above, or something similar, could be encoded in System $\lambda_{\triangleleft}$, then that would constitute a disproof of confluence/commutativity. A direct encoding is not possible because the rules are recursive and require fixpoints. However, we will be adding fixpoints to the language in chapter 3, and they have the following (declarative) subtyping and well-formedness rules:

$$\frac{\Gamma \vdash\ t \leq_{\text{wf}} (\lambda x \leq u.\ u)}{\Gamma \vdash\ (\mathsf{fix}\, t) \leq u} \qquad \frac{\Gamma \vdash\ t \leq_{\text{wf}} (\lambda x \leq u.\ u)}{\Gamma \vdash\ (\mathsf{fix}\, t)\ \text{wf}}$$

These two rules state that the fix operator can be applied to any function for which the body is a subtype of the argument type of the function. Using the fix operator, Bergstra and Klop's counter-example can be encoded as follows. Our encoding uses an inequality rather than an equality, i.e. $C(x) \longrightarrow\!\!\!\!\!\rightarrow E$ iff $C(x) \leq x$, but it still fails the confluence test in the same way. We encode the condition on C as a redex that can only be eliminated if $C(x) \leq x$:

$$\begin{aligned} C &= \mathsf{fix}\ \lambda z \leq \mathsf{Top} \to \mathsf{Top}.\ \lambda x \leq \mathsf{Top}.\ (\lambda y \leq x.\ E)(z(x)) \\ B &= \mathsf{fix}\ \lambda z \leq \mathsf{Top}.\ C(z) \end{aligned}$$

The most important thing to notice about this counter-example is the fact that the body of C is not well-formed, because $z(x)$ is not a subtype of x. We could make $z(x) \leq x$ by changing the type bound of z to $\lambda y' \leq \mathsf{Top}.\ y'$. But if we did that, then our use of the fix operator would not be well-formed, because fix requires a function with a body that is a subtype of the argument type.

We argue that not only is this counter-example ill-formed, but any similar counter-example will be ill-formed. The counter-example crucially depends on having a condition that can fail. If the rewrite condition always succeeds, then it can be removed, thus yielding an ordinary abstract reduction system that is confluent.

In the case of System $\lambda_{\lhd}$, if a redex is well-formed, then the premise (i.e. the condition) on (SRE-APP) will always succeed. Indeed, the whole purpose of the well-formedness judgement is to ensure that functions are always called with arguments of the correct type. If we restrict the subtype relation to well-formed terms, then the condition on β-reduction is unnecessary, and it cannot cause any confluence problems.

2.6.3.2 Join conditions versus equality conditions

However, as described in Section 2.6.2, we have chosen *not* to place a well-formedness requirement on terms in the algorithmic definition of subtyping, because such a requirement would cause a different circular dependency. As Bergstra and Klop show in [Bergstra and Klop, 1986], there is another way to avoid potential problems.

Not only does the counter-example require a condition that can fail, it requires a condition that is *unstable* with respect to reduction. In other words, the condition must be true for some t and u, and false for some t' and u, where $t \longrightarrow\!\!\!\!\!\rightarrow t'$.

A *join condition* is one which relies on t and u having a common reduct: $t = u$ if and only if $t \longrightarrow\!\!\!\!\!\rightarrow \cdot \twoheadleftarrow\!\!\!\!\!\leftarrow u$. In general, join conditions are unstable with respect to reduction. As the counter-example shows, $B \longrightarrow C(B)$, but $E \not\longrightarrow C(E)$, even though $B \longrightarrow\!\!\!\!\!\rightarrow E$. However, if the condition is defined as the symmetric and transitive closure of equality, then the condition is stable by definition; we have $E = C(E)$ because $B = C(B)$ and $B = E$, and the confluence diagram can be completed.

For this reason, the premise of rule (SRE-APP) is given as $s \leq^* t$, rather than $s \leq t$, where $\leq^*$ is transitive closure of $\leq$.

2.6.4 Basic meta-theory of algorithmic subtyping

We now proceed to prove some basic meta-theoretic properties of algorithmic subtyping, much as we did for declarative subtyping. Note that proofs of the two most important properties, namely narrowing and substitution, are only given for $\lhd^*$, not for $\lhd$. As mentioned previously, our goal is to eventually show that transitivity is admissible, which would make $\lhd^*$ and $\lhd$ equivalent relations.

Lemma 2.6.7 (Prevalidity)

If $\Gamma \vdash_A t \lhd u$ then Γ prevalid.
If $\Gamma \vdash_A t \xrightarrow{\lhd} u$ then Γ prevalid.

Proof: By induction on derivations. □

Lemma 2.6.8 (Free variables (algorithmic))

If $x_1 \leq t_1, ..x_n \leq t_n$ prevalid, then every $x_i^{i \in 1..n}$ is distinct.

Proof: By induction on the derivation of prevalidity. □

Lemma 2.6.9 (Weakening)

If $\Gamma, \Gamma' \vdash_A t \leq u$ and $\Gamma, x \leq s, \Gamma'$ prevalid, then $\Gamma, x \leq s, \Gamma' \vdash_A t \leq u$.

Proof: By induction on $t \leq u$. □

Lemma 2.6.10 (Congruence: reduction)

Let $C_{\leq}$ represent a context with a hole in a covariant (positive) position, i.e:
$C_{\leq} ::= [] \mid C_{\leq}(t) \mid \lambda x \leq t.\, C_{\leq}$.
Let $C_{\equiv}$ represent a context with a hole in any position.

If $\Gamma \vdash_A t \xrightarrow{\lhd} u$ then $\Gamma \vdash_A C_{\lhd}[t] \xrightarrow{\lhd} C_{\lhd}[u]$,

Proof: By induction on $C_{\lhd}$, using rules (SR-CONG) and (SR-FUN). □

Lemma 2.6.11 (Congruence: subtyping)

If $\Gamma \vdash_A t \lhd u$ then $\Gamma \vdash_A C_{\lhd}[t] \lhd C_{\lhd}[u]$,

Proof: By induction on $t \lhd u$.
Case (AS-LEFT) and (AS-RIGHT): using lemma 2.6.10 and the inductive hypothesis.
□

Lemma 2.6.12 (Narrowing)

If $\Gamma, x \leq a, \Gamma' \vdash_A t \lhd^* u$ and $\Gamma \vdash_A c \leq^* a$ then $\Gamma, x \leq c, \Gamma' \vdash_A t \lhd^* u$.

Proof: By induction on $t \lhd^* u$, using lemma 2.6.11 (congruence). Every occurence of (AS-LEFT) which promotes x, e.g. $C_{\leq}[x] \xrightarrow{\leq} C_{\leq}[a] \leq^* s$, is replaced with an occurence of (AS-LEFT) and (AST-TRANS), e.g. $C_{\leq}[x] \xrightarrow{\leq} C_{\leq}[c] \leq^* C_{\leq}[a] \leq^* s$. □

Lemma 2.6.13 (Substitution)

If $\Gamma, x \leq a, \Gamma' \vdash_A\ t \lhd^* u$ and $\Gamma \vdash_A\ c \leq^* a$ then $\Gamma, [x \mapsto c]\Gamma' \vdash_A\ [x \mapsto c]t \lhd^* [x \mapsto c]u$.

Proof: By induction on $t \lhd^* u$, using lemma 2.6.11 (congruence). Every occurence of (AS-LEFT) which promotes x, e.g. $C_{\leq}[x] \xrightarrow{\leq} C_{\leq}[a] \leq^* s$, is replaced with an occurence of (AST-TRANS), e.g. $C_{\leq}[c] \leq^* C_{\leq}[a] \leq^* [x \mapsto c]s$. Every reduction of the form $C_{\leq}[y] \xrightarrow{\leq} C_{\leq}[b]$, where $y \in \mathrm{dom}(\Gamma')$, becomes $C_{\leq}[y] \xrightarrow{\leq} C_{\leq}[[x \mapsto c]b]$. □

2.6.5 Equivalence of declarative and algorithmic subtyping.

The declarative subtype relation is only defined over well-formed terms, whereas the algorithmic subtype relation is defined over all terms. Thus, it is not possible to prove complete equivalence between the two systems. However, we can prove that the two systems are equivalent with respect to the set of well-formed terms.

We show that if $\xrightarrow{\equiv}$ and $\xrightarrow{\leq}$ commute, then every well-subtyping rule in the declarative system is admissible in the algorithmic system. We further show that every rule in the algorithmic system is admissible in the declarative system, so long as subtype reductions do not introduce ill-formed terms. This is not property that holds for subtype reduction in general. However, we can ensure that no ill-formed terms will be produced by choosing an appropriate reduction strategy; such strategies are the subject of Section 2.8

2.6.5.1 Declarative subtype rules are admissible in the algorithmic system.

Lemma 2.6.14 (Well-formedness implies prevalidity) If Γ wf then Γ prevalid

Proof: By induction on Γ wf. □

Lemma 2.6.15 (Symmetry is admissible.)

Rule (DS-SYM) is admissible:

$$\frac{\Gamma \vdash_A\ u \equiv t}{\Gamma \vdash_A\ t \equiv u}$$

Proof: By induction on derivations. By lemma 2.6.4 (inversion of subtyping), a derivation of $\Gamma \vdash_A\ u \equiv t$ implies that $\Gamma \vdash_A\ u \xrightarrow{\equiv}\!\!\!\!\rightarrow s \xleftarrow{\equiv} t$ for some s. Swapping things around, we have $\Gamma \vdash_A\ t \xrightarrow{\equiv}\!\!\!\!\rightarrow s \xleftarrow{\equiv} u$, which is a proof that $\Gamma \vdash_A\ t \equiv u$. □

Lemma 2.6.16 (Transitivity is admissible.)

Rule (DS-TRANS) is admissible.

Proof: By lemma 2.6.6 (transitivity elimination). □

Lemma 2.6.17 ($\equiv$ implies $\leq$.)

Rule (DS-EQ) is admissible:

$$\frac{\Gamma \vdash_A \ t \equiv u}{\Gamma \vdash_A \ t \leq u}$$

Proof: By induction on derivations. By lemma 2.6.4 (inversion of subtyping), a transitivity-free derivation of $\Gamma \vdash_A \ t \equiv u$ implies that $\Gamma \vdash_A \ t \xrightarrow{\equiv}\!\!\!\!\!\rightarrow s \xleftarrow{\equiv} u$ for some s. Using rule (SR-EQ), $t \xrightarrow{\equiv}\!\!\!\!\!\rightarrow s$ implies $t \xrightarrow{\leq}\!\!\!\!\!\rightarrow s$, thus giving us $\Gamma \vdash_A \ t \leq u$. □

Lemma 2.6.18 (Congruence rules are admissible.)

Rules (DS-VAR), (DS-TOP), (DS-APP), and (DS-FUN) are admissible if Γ prevalid.

$$\Gamma \vdash_A \ x \equiv x \quad \text{(DS-VAR)} \qquad \Gamma \vdash_A \ \mathsf{Top} \equiv \mathsf{Top} \quad \text{(DS-TOP)}$$

$$\frac{\Gamma \vdash_A \ t \equiv t' \qquad \Gamma, x \leq t \vdash_A \ u \lhd u'}{\Gamma \vdash_A \ \lambda x \leq t.\ u \ \lhd \ \lambda x \leq t'.\ u'} \quad \text{(DS-FUN)} \qquad \frac{\Gamma \vdash_A \ t \lhd t', \quad u \equiv u'}{\Gamma \vdash_A \ t(u) \lhd t'(u')} \quad \text{(DS-APP)}$$

Proof: (DS-VAR) and (DS-TOP) follow from (AS-REFL). (DS-APP) and (DS-FUN), follow using a similar logic to lemma 2.6.11 (congruence). □

Lemma 2.6.19 (DS-EVAR and DS-EAPP are admissible.)

Rules (DS-EVAR) and (DS-EAPP) are admissible, if Γ prevalid.

$$\Gamma \vdash_A \ t \leq \mathsf{Top} \quad \text{(DS-ETOP)} \qquad \frac{x \leq t \in \Gamma}{\Gamma \vdash_A \ x \leq t} \quad \text{(DS-EVAR)}$$

Proof: (DS-EVAR) is admissible using (SRS-VAR), and (DS-ETOP) using (SRS-TOP). □

Theorem 2.6.20 (Declarative subtyping implies algorithmic subtyping)

If $\Gamma \vdash \ t \leq_{\text{wf}} u$, then $\Gamma \vdash_A \ t \leq u$, assuming that $\xrightarrow{\leq}$ and $\xrightarrow{\equiv}$ commute.

Proof: By induction on the derivation of $t \leq_{\text{wf}} u$.

By lemma 2.5.3 (well-definedness), a well-subtyping judgement will only compare terms that are well-formed. By lemma 2.5.1 (context formation), we know that Γ wf, and therefore Γ prevalid. All subtyping judgements in the declarative system are admissible in the algorithmic system according the lemmas shown above, with the exception of (DS-EAPP), which we consider below:

$$\Gamma \vdash_A \ (\lambda x \leq a.\ b)(c) \equiv [x \mapsto c]b \quad \text{(DS-EAPP)}$$

(DS-EAPP) is admissible using (SRE-APP), provided that $\Gamma \vdash_A \ c \leq^* a$. Since $\Gamma \vdash \ (\lambda x \leq a.\ b)(c)$ wf, we obtain $\Gamma \vdash_A \ (\lambda x \leq a.\ b) \leq (\lambda x \leq a'.\ \mathsf{Top})$ by the induction hypothesis. We then derive $\Gamma \vdash_A \ a \equiv a'$ using lemma 2.6.4 (inversion of subtyping), and obtain $\Gamma \vdash_A \ c \leq^* a$ using rule (AST-TRANS). □

2.6.5.2 Algorithmic rules are admissible in the declarative system

Lemma 2.6.21 ($\Gamma \vdash \ t \lhd t$ in the declarative system)

Rule (AS-REFL) is admissible in the declarative system.

$$\frac{\Gamma \text{ prevalid}}{\Gamma \vdash \ t \lhd t}$$

Proof: By induction on t. □

Lemma 2.6.22 ($\xrightarrow{\lhd}$ implies $\lhd$)

If $\Gamma \vdash \ t$ wf, and $\Gamma \vdash_A \ t \xrightarrow{\lhd} t'$, then $\Gamma \vdash \ t \leq t'$.

Proof: By induction on the derivation of $\Gamma \vdash_A \ t \xrightarrow{\lhd} t'$. Case (SRE-APP): by (DS-EAPP).
Case (SRE-TOPAPP): Can't happen; $\mathsf{Top}(t)$ is not well-formed.
Case (SRS-TOP): by (DS-ETOP).
Case (SRS-VAR): by (DS-EVAR).
Case (SR-FUN): by induction, using (DS-FUN).
Case (SR-CONG): by induction, using (DS-APP) and (DS-FUN). □

Theorem 2.6.23 (Algorithmic subtyping implies declarative subtyping)

If $\Gamma \vdash_A \ t \leq u$ is a derivation in the algorithmic system that only includes well-formed terms, then $\Gamma \vdash \ t \leq u$ in the declarative system.

Proof: By induction on $\Gamma \vdash_A t \leq u$.
Case (AS-REFL): by lemma 2.6.21.
Case (AS-LEFT), $t \xrightarrow{\leq} t' \leq u$: by lemma 2.6.22 and (DS-TRANS).
Case (AS-LEFT), $t \leq u' \xleftarrow{\equiv} u$: by lemma 2.6.22, (DS-TRANS) and (DS-SYM).

Note that rule (DS-TRANS) requires intermediate terms to be well-formed. However, $\Gamma \vdash t$ wf, and $\Gamma \vdash_A t \xrightarrow{\leq} t'$ does not guarantee that t' is well-formed; this issue is discussed in more detail in Section 2.8.4. Algorithmic subtype derivations are therefore admissible only if they do not introduce ill-formed terms.

□

2.7 Confluence and commutativity

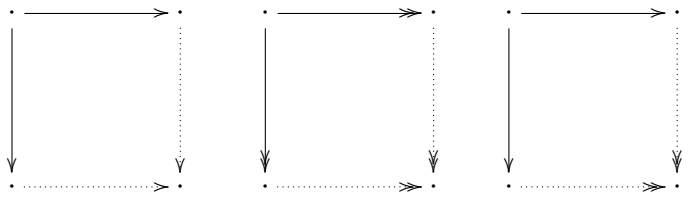
Thus far, we have shown that System $\lambda_{\lhd}$ is sound if declarative subtyping has the transitivity elimination property. Transitivity elimination is very difficult to prove in the declarative formulation of subtyping, so we introduced an algorithmic subtype relation, and showed that the declarative system and the algorithmic systems are equivalent. We further showed that transitivity is admissible in the algorithmic system if $\xrightarrow{\equiv}$ and $\xrightarrow{\leq}$ commute.

This section discusses the question of commutativity in more detail, and it is here that we run into trouble. We can show that $\xrightarrow{\equiv}$ and $\xrightarrow{\leq}$ commute locally, but we have thus far been unable to show that they commute globally.

2.7.1 An overview of confluence properties and proofs

To better explain the problem, we give a brief overview of confluence properties and proofs. As we mentioned earlier, confluence and commutativity are very similar properties. The proof techniques are essentially the same in both cases, so the following discussion applies to both. All examples and proofs in this section are drawn from [Bezem et al., 2003].

There are actually three different properties related to confluence, which are illustrated by the following diagram.

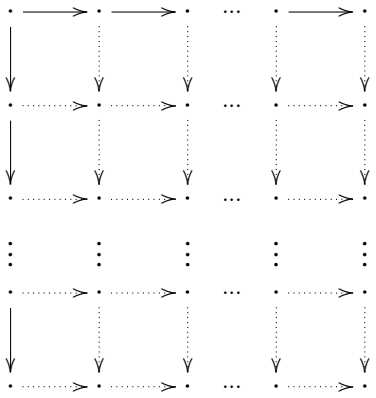


diamond property global confluence local confluence

For the diamond property, both the spanning and completing edges consist of a single reduction. Global confluence, which is otherwise known simply as "confluence", has multiple reductions on both the spanning and completing edges. Local confluence is a cross between the two; it has a single reduction on the spanning edges, but multiple reductions on the completing edges. These three properties are not equivalent; they are positioned from left to right in order of strength. The diamond property is strictly stronger than global confluence, and global confluence is strictly stronger than local confluence.

Although confluence is the most important of the three properties, it is usually not possible to prove it directly, because the spanning edges of a confluence diagram consist of an arbitrary number of unknown reductions. Instead, most proofs start by showing either the diamond property or local confluence. The spanning edges in those two cases consist of just a single reduction, so the proof is by simple analysis of all possible combinations.

The diamond property implies confluence because it can be used to *tile* a diagram, as shown below:

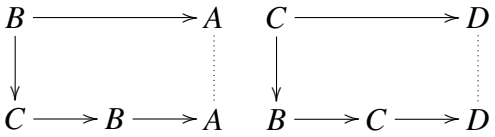


Since the diamond property is strictly stronger than confluence, there are systems which are confluent but do not have a diamond property. If there is no diamond property, then proving confluence is considerably more difficult. A common way to ap-

proach the proof is to show local confluence instead. However, local confluence does not imply global confluence. The standard counter-example is the following reduction system, which has four terms: A, B, C, and D.

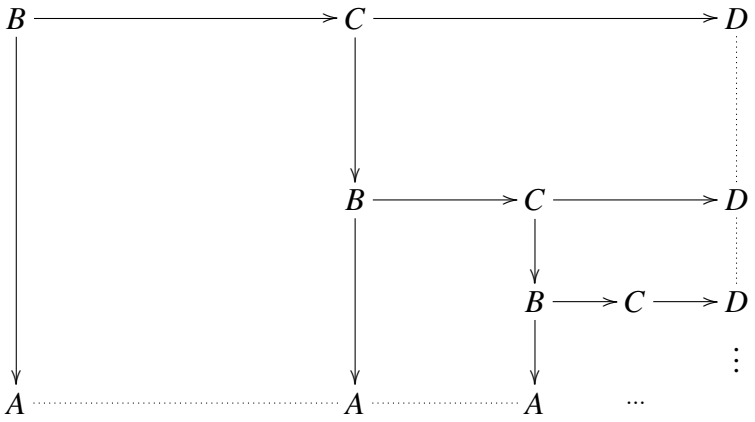
$$\begin{array}{lll} B & \longrightarrow & A \\ C & \longrightarrow & B \end{array} \qquad \begin{array}{lll} B & \longrightarrow & C \\ C & \longrightarrow & D \end{array}$$

It is easy to see that this reduction system is locally confluent, as shown by the following diagrams:



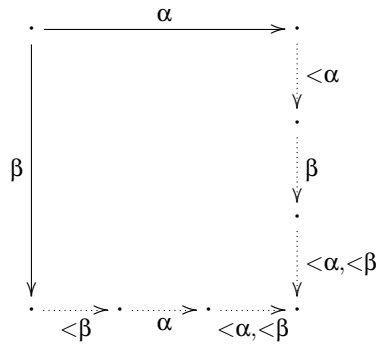
However, this system is not globally confluent. Consider the reductions $B \longrightarrow A$, and $B \longrightarrow C \longrightarrow D$. If the system were confluent, then we should have $A \longrightarrow\!\!\!\!\!\rightarrow \cdot \longleftarrow\!\!\!\!\!\leftarrow D$. However, A and D are normal forms, so they have no common reduct.

It is possible to use local confluence diagrams as tiles to fill in a global confluence diagram, in much the same way as we did with the diamond property. However, unlike the diamond property, the tiling process is not guaranteed to terminate. Such non-termination causes confluence to fail, e.g.



In order to prove global confluence from local confluence, an additional induction principle is needed to ensure that the tiling process can be completed. The induction principle most commonly used in the literature is strong normalization, which induces a well-founded order on terms. According to Newman's lemma, a reduction system is globally confluent if it is both locally confluent and strongly normalizing [Bezem et al., 2003].

Alternatively, one can use a well-founded order on reductions. According to Van Oostrom's technique of decreasing diagrams [van Oostrom, 1994] [Klop et al., 2000], a reduction system is confluent if the diagrams of local confluence have the following form:



Diagrams of this form are called *elementary decreasing diagrams*. Each reduction is assigned an index, represented by α and β, and there is a well-founded order on indices. A completing edge is allowed to have multiple reductions, but only one reduction can have the same index as the spanning edge. Because all of the additional reductions have a smaller index, it is possible to show that the tiling process terminates.

In System $\lambda_{\triangleleft}$, the $\xrightarrow{\equiv}$ relation has a diamond property, and is therefore confluent. However, the commuting diagrams for $\xrightarrow{\leq}$ and $\xrightarrow{\equiv}$ do not have a diamond property; they only commute locally. System $\lambda_{\triangleleft}$ is not strongly normalizing, so we will look for a possible way to assign indices to reductions.

2.7.2 Simultaneous reduction

As is standard practice in confluence proofs, we begin by defining a notion of simultaneous reduction [Takahashi, 1995] [Bezem et al., 2003]. Simultaneous reduction, written $t \xrightarrow{\equiv}\!\!\!\!\!\!\circ\ t'$, allows multiple subterms to be reduced in parallel. The $\xrightarrow{\equiv}\!\!\!\!\!\!\circ$ relation is defined in Figure 2.8.

Our presentation is very similar to [Takahashi, 1995]. However, Takahashi defines simultaneous reduction for the untyped λ-calculus. In our presentation, $\xrightarrow{\equiv}\!\!\!\!\!\!\circ$ is defined within a type context Γ, and rule (CR-BETA) has a premise $\Gamma \vdash s' \leq^* t$ that does not exist in the usual formulation. We show that the basic properties of simultaneous reduction are unaffected by these changes.

Lemma 2.7.1 ($\xrightarrow{\equiv}\!\!\!\!\!\!\circ$ is reflexive)
$\Gamma \vdash_A t \xrightarrow{\equiv}\!\!\!\!\!\!\circ\ t$

$$\frac{\Gamma \text{ prevalid}}{\Gamma \vdash_A\ x \xrightarrow[\circ]{\equiv} x} \quad \text{(CR-VAR)}$$

$$\frac{\Gamma \text{ prevalid}}{\Gamma \vdash_A\ \mathsf{Top} \xrightarrow[\circ]{\equiv} \mathsf{Top}} \quad \text{(CR-TOP)}$$

$$\frac{\Gamma \vdash_A\ t \xrightarrow[\circ]{\equiv} t', \quad u \xrightarrow[\circ]{\equiv} u'}{\Gamma \vdash_A\ t(u) \xrightarrow[\circ]{\equiv} t'(u')} \quad \text{(CR-APP)}$$

$$\frac{\Gamma \vdash_A\ t \xrightarrow[\circ]{\equiv} t' \qquad \Gamma, x \le t' \vdash_A\ u \xrightarrow[\circ]{\equiv} u'}{\Gamma \vdash_A\ \lambda x \le t.\ u \xrightarrow[\circ]{\equiv} \lambda x \le t'.\ u'} \quad \text{(CR-FUN)}$$

$$\frac{\Gamma \vdash_A\ s \xrightarrow[\circ]{\equiv} s', \quad s' \le^* t \qquad \Gamma,\ x \le t \vdash_A\ u \xrightarrow[\circ]{\equiv} u'}{\Gamma \vdash_A\ (\lambda x \le t.\ u)(s) \xrightarrow[\circ]{\equiv} [x \mapsto s']u'} \quad \text{(CR-BETA)}$$

$$\frac{\Gamma \text{ prevalid}}{\Gamma \vdash_A\ \mathsf{Top}(t) \xrightarrow[\circ]{\equiv} \mathsf{Top}} \quad \text{(CR-TOPAPP)}$$

Figure 2.8: Simultaneous reduction

Proof: By induction on the size of t. □

Lemma 2.7.2 ($\xrightarrow{\equiv} \subseteq \xrightarrow[\circ]{\equiv} \subseteq \xrightarrow{\equiv}\!\!\!\!\rightarrow$)

(I) If $\Gamma \vdash_A\ t \xrightarrow{\equiv} u$ then $\Gamma \vdash_A\ t \xrightarrow[\circ]{\equiv} u$.

(II) If $\Gamma \vdash_A\ t \xrightarrow[\circ]{\equiv} u$ then $\Gamma \vdash_A\ t \xrightarrow{\equiv}\!\!\!\!\rightarrow u$.

Proof:
Part (I) follows by induction on $t \xrightarrow{\equiv} u$, and lemma 2.7.1.
Part (II) follows from induction on $t \xrightarrow[\circ]{\equiv} u$. □

Lemma 2.7.3 (Substitution over $\xrightarrow[\circ]{\equiv}$)

If $\Gamma, x \le s, \Gamma' \vdash_A\ t \xrightarrow[\circ]{\equiv} t'$ and $\Gamma \vdash_A\ u \xrightarrow[\circ]{\equiv} u' \le s$ then $\Gamma, [x \mapsto u']\Gamma' \vdash_A\ [x \mapsto u]t \xrightarrow[\circ]{\equiv} [x \mapsto u']t'$.

Proof: By induction on $t \xrightarrow[\circ]{\equiv} t'$. We have the following cases:

$x \xrightarrow[\circ]{\equiv} x$	becomes $u \xrightarrow[\circ]{\equiv} u'$
$y \xrightarrow[\circ]{\equiv} y$ s.t. $y \neq x$	trivial
$\mathsf{Top} \xrightarrow[\circ]{\equiv} \mathsf{Top}$	trivial
$\mathsf{Top}(t) \xrightarrow[\circ]{\equiv} \mathsf{Top}$	trivial
$a(b) \xrightarrow[\circ]{\equiv} a'(b')$	by induction on $a \xrightarrow[\circ]{\equiv} a'$ and $b \xrightarrow[\circ]{\equiv} b'$.
$\lambda y \le a.\ b \xrightarrow[\circ]{\equiv} \lambda y \le a'.\ b'$	by induction on $a \xrightarrow[\circ]{\equiv} a'$ and $b \xrightarrow[\circ]{\equiv} b'$.
$(\lambda y \le a.\ b)(c) \xrightarrow[\circ]{\equiv} [y \mapsto c']b'$	by induction on $b \xrightarrow[\circ]{\equiv} b'$ and $c \xrightarrow[\circ]{\equiv} c'$.

For last case, we must also show the premise of (CR-Beta) is satisfied, i.e. $\Gamma, [x \mapsto u']\Gamma' \vdash_A [x \mapsto u']c' \leq^* [x \mapsto u]a$. We have $\Gamma, [x \mapsto u']\Gamma' \vdash_A [x \mapsto u']c' \leq^* [x \mapsto u']a$, by 2.6.13 (substitution over $c' \leq a$). The premise is satisfied using AS-RIGHT, along with the fact that $u \xrightarrow{\equiv}\!\!\!\!\!\rightarrow u'$. □

Definition 2.7.4 ($\Gamma \xrightarrow[\circ]{\equiv} \Gamma'$) $\Gamma \xrightarrow[\circ]{\equiv} \Gamma'$ is defined as follows:

$$\emptyset \xrightarrow[\circ]{\equiv} \emptyset \qquad \frac{\Gamma \xrightarrow[\circ]{\equiv} \Gamma' \quad \Gamma' \vdash_A a \xrightarrow[\circ]{\equiv} a'}{\Gamma, x \leq a \xrightarrow[\circ]{\equiv} \Gamma', x \leq a'}$$

Lemma 2.7.5 (Change of context)

(I) If $\Gamma \xrightarrow[\circ]{\equiv} \Gamma'$, and $\Gamma' \vdash_A t \xrightarrow[\circ]{\equiv} u$, then $\Gamma \vdash_A t \xrightarrow[\circ]{\equiv} u$.
(II) If $\Gamma \xrightarrow[\circ]{\equiv} \Gamma'$, and $\Gamma \vdash_A t \xrightarrow[\circ]{\equiv} u$, then $\Gamma' \vdash_A t \xrightarrow[\circ]{\equiv} u$.

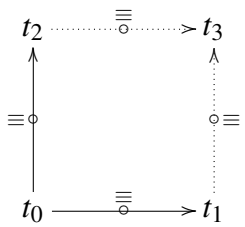
Proof: The proof for both cases is similar, by induction on $t \xrightarrow[\circ]{\equiv} u$. The only rule that is affected by context is CR-BETA, which has a premise: $(\lambda x \leq a.b)(c) \xrightarrow[\circ]{\equiv} [x \mapsto c']b'$ only if $c' \leq^* a$. A subtype derivation in Γ' holds in Γ, or vice versa, by lemma 2.6.12 (narrowing). □

2.7.3 Confluence of $\xrightarrow{\equiv}$

Our proof of conflunce for $\xrightarrow{\equiv}$ follows the standard proof of confluence for the untyped λ-calculus. The version here is a modification of Takahashi's proof [Takahashi, 1995], which is also described in [Bezem et al., 2003].

Lemma 2.7.6 ($\xrightarrow[\circ]{\equiv}$ has the Diamond Property.)

If $\Gamma \vdash_A t_0 \xrightarrow[\circ]{\equiv} t_1$, $t_0 \xrightarrow[\circ]{\equiv} t_2$ then there exists a t_3, such that $\Gamma \vdash_A t_1 \xrightarrow[\circ]{\equiv} t_3$, $t_2 \xrightarrow[\circ]{\equiv} t_3$, as illustrated by the following diagram:



Proof: By induction on $t_0 \xrightarrow[\circ]{\equiv} t_1$. The various cases are shown in Figure 2.9. Each case is defined inductively, so if term t_0 contains a_0 as a subterm, then $a_1 \xrightarrow[\circ]{\equiv} a_3$ and

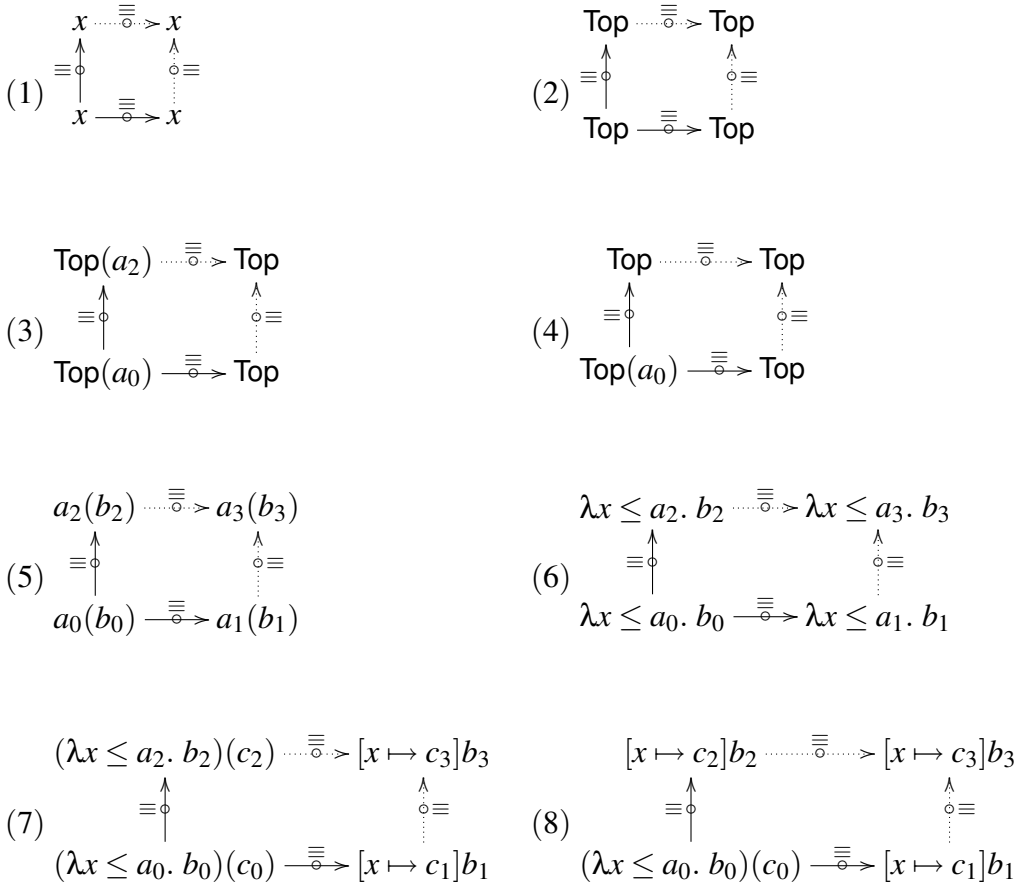


Figure 2.9: Elementary confluence diagrams for $\xrightarrow{\equiv}$

$a_2 \xrightarrow{\equiv} a_3$ are by induction on $a_0 \xrightarrow{\equiv} a_1$ and $a_0 \xrightarrow{\equiv} a_2$, and similarly for b_0 and c_0.

In case (6) the spanning edges have subderivations $\Gamma, x \leq a_1 \vdash\ b_1 \xrightarrow{\equiv} b_3$ and $\Gamma, x \leq a_2 \vdash\ b_2 \xrightarrow{\equiv} b_3$. We adjust the context in both cases to $\Gamma, x \leq a_3$ using lemma 2.7.5 (Change of Context), and then apply the induction hypothesis. Case (7) is similar.

Case (7) employs lemma 2.7.3 (Substitution over $\xrightarrow{\equiv}$) to derive $[x \mapsto c_1]b_1 \xrightarrow{\equiv} [x \mapsto c_3]b_3$. $\square$

Theorem 2.7.7 (The $\xrightarrow{\equiv}$ relation is confluent)

If $\Gamma \vdash_A\ t_0 \xrightarrow{\equiv\!\!\!\rightarrow} t_1,\ t_0 \xrightarrow{\equiv\!\!\!\rightarrow} t_2$ then there exists a t_3, such that $\Gamma \vdash_A\ t_1 \xrightarrow{\equiv\!\!\!\rightarrow} t_3,\ t_2 \xrightarrow{\equiv\!\!\!\rightarrow} t_3$

Proof: By lemma 2.7.2 ($\xrightarrow{\equiv} \subseteq \xrightarrow{\equiv}_{\circ} \subseteq \xrightarrow{\equiv}$), the two spanning edges are rewritten as $t_0 \xrightarrow{\equiv}_{\circ} t_1$ and $t_0 \xrightarrow{\equiv}_{\circ} t_2$. The diagram is then tiled using lemma 2.7.6. □

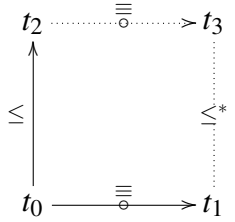
2.7.4 Commutativity of $\xrightarrow{\leq}$ and $\xrightarrow{\equiv}$

As mentioned earlier, $\xrightarrow{\leq}$ and $\xrightarrow{\equiv}$ do not have a diamond property, so we can only show that they commute locally. The actual property is not even local commutativity per se, but something quite similar:

Lemma 2.7.8 (Weak commutativity of $\xrightarrow{\leq}$ and $\xrightarrow{\equiv}$)

If $\Gamma \vdash_A t_0 \xrightarrow{\equiv}_{\circ} t_1$, $t_0 \xrightarrow{\leq} t_2$, and $\Gamma \xrightarrow{\equiv}_{\circ} \Gamma'$, then there exists a t_3, such that $\Gamma \vdash_A t_2 \xrightarrow{\equiv}_{\circ} t_3$, and $\Gamma' \vdash_A t_1 \leq^* t_3$.

This property is illustrated by the following diagram. Note that the completing edge of the diagram on the right-hand side is not a subtype reduction (which would yield a diamond property) but a subtyping judgement, which we represent by drawing a dotted line instead of an arrow. Note also that the right edge is done in a different type context (Γ') than the left edge.



Proof: By induction on $t_0 \xrightarrow{\equiv}_{\circ} t_1$. If $t_0 \xrightarrow{\leq} t_2$ is by rule (SR-EQ), then the result follows from the diamond property of $\xrightarrow{\equiv}_{\circ}$. The other cases are shown in Figure 2.10.

In case (1), the reduction on the left is $\Gamma_0, x \leq a_0, \Gamma_0' \vdash x \xrightarrow{\leq} a_0$, while the one on the right is $\Gamma_1, x \leq a_1, \Gamma_1' \vdash x \xrightarrow{\leq} a_1$, where $\Gamma_0, x \leq a_0, \Gamma_0' \xrightarrow{\equiv}_{\circ} \Gamma_1, x \leq a_1, \Gamma_1'$.

Case (2) does a Top promotion on both left and right.

Cases (3), (4), and (5) are by induction on $b_0 \xrightarrow{\equiv}_{\circ} b_1$ and $b_0 \xrightarrow{\leq} b_2$. The induction hypothesis allows the change in context in case (4).

Case (5) makes use of lemma 2.6.13 (substitution) to obtain $[x \mapsto c_1]b_1 \leq^* [x \mapsto c_2]b_2$. The substitution lemma is applicable because $c_1 \leq^* a_0$ is a premise of rule (CR-BETA).

Case (6) demonstrates why $\mathsf{Top}(t) \xrightarrow{\equiv} \mathsf{Top}$ must be included as a reduction in the algorithmic system, even though it is not well-formed. □

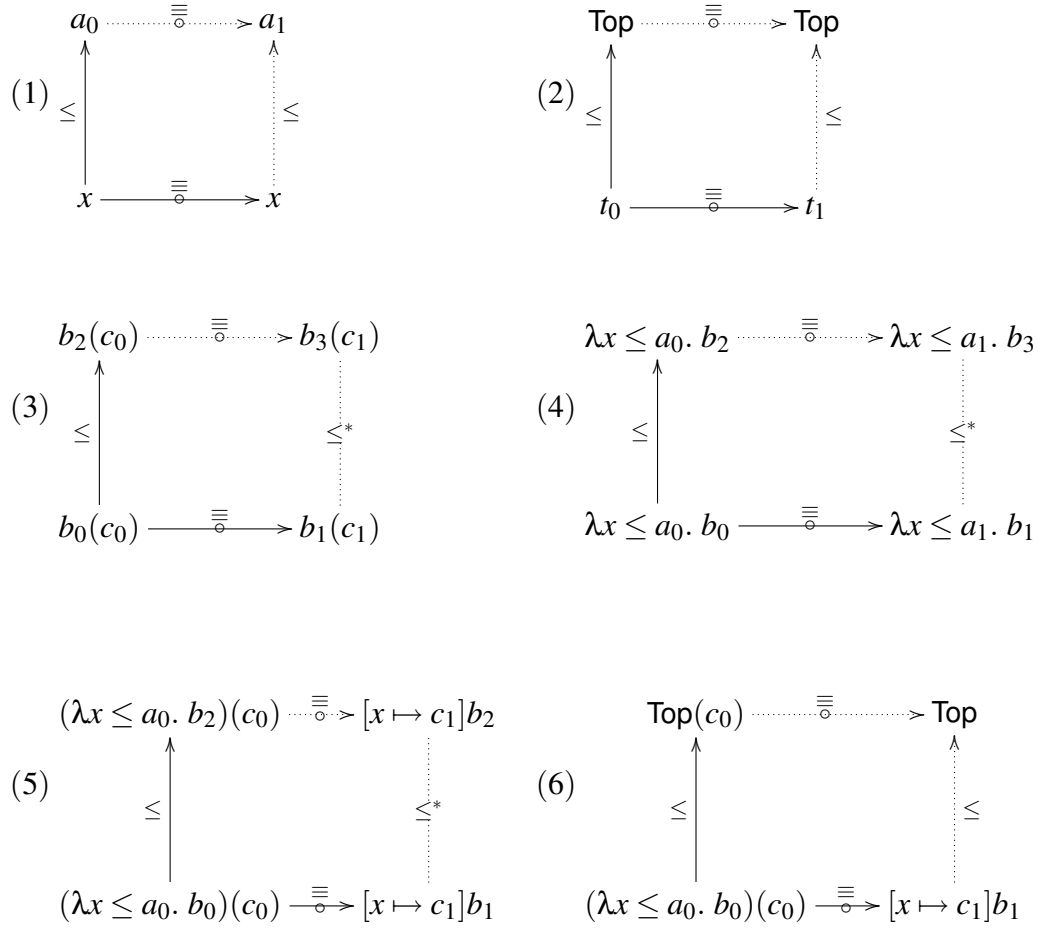
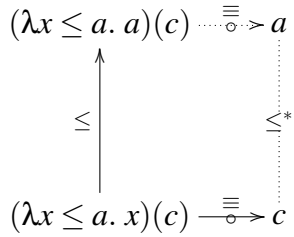


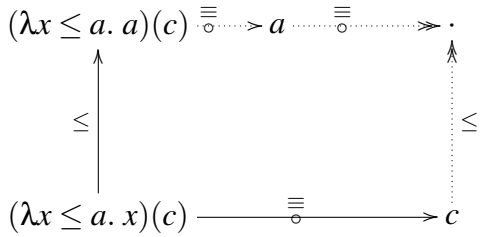
Figure 2.10: Elementary diagrams for commutativity of $\xrightarrow{\leq}$ and $\xrightarrow{\equiv}$

Case (5) is the most interesting of the six cases shown above, because it is the case that causes the diamond property to fail. Cases (1), (2) and (6) are simple commuting diamonds; the completing edge on the right side of each diagram is a single subtype reduction. Cases (3) and (4) are by induction, and thus follow the pattern dictated by the induction hypothesis; these would be diamonds if case (5) was a diamond.

However, case (5) is not a diamond. The subtype reduction from $b_0 \xrightarrow{\leq} b_2$ may promote x to a_0. If it does, then a single reduction from $x \xrightarrow{\leq} a_0$ on the left edge of the diagram will be replaced with the subtype judgement $c_1 \leq^* a_0$ on the right edge of the diagram. For example:



If $c \leq^* a$ can be converted to $c \leq a$, (perhaps by using transitivity elimination as an induction hypothesis), then this case corresponds to a diagram of local commutativity, e.g.



2.7.5 Decreasing diagrams

Lemma 2.7.8 (weak commutativity), implies full commutativity if the following two conditions hold:

1. The subtype judgement on the right completing edge has a transitivity-free derivation.
2. The indices of the reductions in the transitivity-free derivation are strictly smaller than those on the spanning edges (confluence by decreasing diagrams).

These two conditions are related. If the subtype derivation on the right edge only contains reductions with smaller indices, then the first condition can be satisfied by using commutativity as an induction hypothesis, along with lemma 2.6.6 (transitivity elimination). Van Oostrom's decreasing diagrams technique guarantees that indices never increase, so the second condition will be satisfied as well; the indices of the transitivity-free derivation must be less than or equal to the the indices of the original.

The subtype judgement on the right edge is, in fact, smaller than the spanning reductions by the definition of depth shown in Figure 2.11. The depth function is defined over derivations of subtype reduction ($\xrightarrow{\leq}$), parallel reduction ($\xrightarrow{\equiv}$), algorithmic subtyping ($\leq$), and transitive subtyping ($\leq^*$), and it yields a natural number. (Parallel reduction is used in place of equivalence reduction ($\xrightarrow{\equiv}$).) The definition states that

$$
\begin{array}{lll}
\mathrm{depth}(t \lhd t) \quad (\text{AS-REFL}) & = & 0 \\
\mathrm{depth}(t \xrightarrow{\lhd} t' \lhd u) \quad (\text{AS-LEFT}) & = & \max(\mathrm{depth}(t \xrightarrow{\lhd} t'), \mathrm{depth}(t' \lhd u)) \\
\mathrm{depth}(t \leq u',\ u \xrightarrow{\equiv} u') \quad (\text{AS-RIGHT}) & = & \max(\mathrm{depth}(t \leq u'), \mathrm{depth}(u \xrightarrow{\equiv} u')) \\
\mathrm{depth}(s \leq^* t,\ t \leq^* u) \quad (\text{AST-TRANS}) & = & \max(\mathrm{depth}(s \leq^* t), \mathrm{depth}(t \leq^* u))
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{depth}(x \xrightarrow{\leq} t) & = & 0 \\
\mathrm{depth}(t \xrightarrow{\leq} \mathsf{Top}) & = & 0 \\
\mathrm{depth}((\lambda x \leq t.\ u) \xrightarrow{\leq} (\lambda x \leq t.\ u')) & = & \mathrm{depth}(u \xrightarrow{\leq} u') \\
\mathrm{depth}(t(u) \xrightarrow{\leq} t'(u)) & = & \mathrm{depth}(t \xrightarrow{\leq} t')
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{depth}(\mathsf{Top} \xrightarrow{\equiv\!\!\circ\!\!\rightarrow} \mathsf{Top}) & = & 0 \\
\mathrm{depth}(x \xrightarrow{\equiv\!\!\circ\!\!\rightarrow} x) & = & 0 \\
\mathrm{depth}(\mathsf{Top}(t) \xrightarrow{\equiv\!\!\circ\!\!\rightarrow} \mathsf{Top}) & = & 0 \\
\mathrm{depth}(\lambda x \leq t.\ u \xrightarrow{\equiv\!\!\circ\!\!\rightarrow} \lambda x \leq t'.\ u') & = & \max(\mathrm{depth}(t \xrightarrow{\equiv\!\!\circ\!\!\rightarrow} t'), \mathrm{depth}(u \xrightarrow{\equiv\!\!\circ\!\!\rightarrow} u')) \\
\mathrm{depth}(t(u) \xrightarrow{\equiv\!\!\circ\!\!\rightarrow} t'(u')) & = & \max(\mathrm{depth}(t \xrightarrow{\equiv\!\!\circ\!\!\rightarrow} t'), \mathrm{depth}(u \xrightarrow{\equiv\!\!\circ\!\!\rightarrow} u')) \\
\mathrm{depth}((\lambda x \leq t.\ u)(s) \xrightarrow{\equiv\!\!\circ\!\!\rightarrow} [x \mapsto s']u') & = & \max(\mathrm{depth}(u \xrightarrow{\equiv\!\!\circ\!\!\rightarrow} u'), \mathrm{depth}(s \xrightarrow{\equiv\!\!\circ\!\!\rightarrow} s'), \\
 & & \quad 1 + \mathrm{depth}(s' \leq^* t))
\end{array}
$$

Figure 2.11: The depth of subtyping derivations

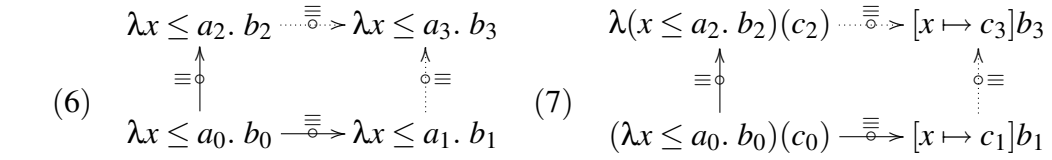
---

the depth of a subtype derivation is the same as the depth of the largest reduction in the derivation. The depth of a β-reduction $(\lambda x \leq a.\ b)(c) \xrightarrow{\equiv} [x \mapsto c]b$ is one plus the depth of $c \leq^* a$. All other reductions have a depth of zero. Min and max are the standard minimum and maximum functions on natural numbers.

Referring back to lemma 2.7.8, every case is an elementary decreasing diagram when reductions are indexed by depth. In case (5), which is the interesting case, the right completing edge is constructed by substituting a subtype judgement that is strictly smaller than the bottom edge.

2.7.6 Failure of decreasing diagrams

The proof that $\xrightarrow{\equiv}$ and $\xrightarrow{\leq}$ commute requires one more step: the confluence diamonds for $\xrightarrow{\equiv\!\!\circ\!\!\rightarrow}$ must also be decreasing with respect to depth. Unfortunately, it is here that the proof breaks down; the diagrams for $\xrightarrow{\equiv\!\!\circ\!\!\rightarrow}$ are not decreasing. The problem is caused by cases (6) and (7):



In Case (6), the spanning edges $b_0 \xrightarrow{\equiv\!\!\circ} b_1$ and $b_0 \xrightarrow{\equiv\!\!\circ} b_2$ are done within the contet $\Gamma, x \leq a_1$, and $\Gamma, x \leq a_2$, respectively. The completing edges $b_1 \xrightarrow{\equiv\!\!\circ} b_3$ and $b_2 \xrightarrow{\equiv\!\!\circ} b_3$ are done within a different context: $\Gamma, x \leq a_3$. Subtype judgements within the old context hold in the new context by lemma 2.6.12 (narrowing). However, this lemma replaces each judgement of the form $x \xrightarrow{\leq} a_2$ with $x \xrightarrow{\leq} a_3 \xleftarrow{\equiv\!\!\circ} a_2$, which increases the depth of the judgement.

Case (7) has a more obvious problem. The bottom edge has premise $c_1 \leq a_0$, while the top edge has premise $c_2 \leq a_2$. The premise of the top edge follows from rule (AST-TRANS), using $c_0 \xrightarrow{\equiv\!\!\circ} c_2$ and $a_0 \xrightarrow{\equiv\!\!\circ} a2$, but it has a different depth. Similarly, the right edge is $[x \mapsto c_1]b_1 \xrightarrow{\equiv\!\!\circ} [x \mapsto c_3]b_3$, and substitution may increase the depth.

2.7.7 Confluence: summary and discussion

The proof of transitivity elimination for System $\lambda_{\lhd}$ has two parts: (1) confluence of $\xrightarrow{\equiv}$, and (2) commutativity of $\xrightarrow{\equiv}$ and $\xrightarrow{\leq}$. When each part is considered separately, there is a proof technique that naturally applies. Confluence of $\xrightarrow{\equiv}$ is by induction on the number of simultaneous reductions (i.e. the diamond property). Commutativity is by induction on the depth of reductions, using the technique of decreasing diagrams.

Unfortunately, we are unable to combine these two induction principles into a single proof. The diagrams for $\xrightarrow{\equiv\!\!\circ}$ do not preserve depth, and the diagrams for commutativity do not preserve the number of simultaneous reductions. A complete proof of commutativity thus requires a stronger induction principle. Strong normalization would provide such a principle, but System $\lambda_{\lhd}$ is not strongly normalizing.

In the absense of a suitable induction principle, it makes sense to look for counter-examples to commutativity. A counter-example would produce an infinite tiling, as described in section 2.7.1, by generating an sequence of subtype derivations that never decrease in size. Constructing such a counter-example in System $\lambda_{\lhd}$ itself would be extremely difficult, because an infinite tiling implies an infinite reduction sequence. The only known infinite reduction sequences arise from Girard's paradox, and involve

terms that are so large as to defy easy analysis (see Section 2.4.4). However, in subsequent sections we will introduce fixpoints to the language, which make the construction of non-terminating expressions much easier.

Nevertheless, even with fixpoints, we have been unable to construct such a counter-example. The partial proof of commutativity given here is "close enough" to a full proof that all of the obvious possibilities can be eliminated. Since each half of the proof has a valid induction principle when considered on its own, any counter-example would have to exploit some subtle interaction between the two halves. As discussed in Section 2.6.3.1, the classic counter-examples from the theory of conditional rewrite systems do not apply either.

[Intrigila et al., 2001] outlines some of the basic characteristics of rewrite systems that are locally but not globally confluent. Such systems have two or more "basins of attraction": sets of terms for which reductions lead in to the set but not away from it. In addition, there must be an infinite reduction sequence that lies along the edge that separates two different basins. There is no reason to suspect that System $\lambda_{\lhd}$ has this kind of structure. However, in the absence of a proof, the possibility cannot be ruled out.

Section 2.9 outlines some of the different ways in which future research could potentially work around the problems with our existing proof, and it discusses some alternative proof techniques that might apply to transitivity elimination. For now however, we will set theory aside for the moment, and show how subtyping can be implemented in practice.

2.8 Practical algorithms for subtyping

The definition of "algorithmic subtyping" given in Figure 2.7, is not actually an algorithm, because it is not entirely syntax directed. To make it into a proper algorithm, we must specify a strategy for performing reductions.

In a compiler, the subtyping algorithm would typically be invoked from the well-formedness judgement, since type-checking involves verifying that a program is well-formed. The well-formedness judgement actually uses subtyping in two distinct ways, both of which can be seen in (W-APP):

$$\frac{\Gamma \vdash\ t \leq_{\mathrm{wf}} (\lambda x \leq s.\ \mathsf{Top}), \quad u \leq_{\mathrm{wf}} s}{\Gamma \vdash\ t(u)\ \mathrm{wf}}$$

The first way of using the subtyping judgement is as a comparison between terms. This is the way that subtyping is usually used; given two terms u and s, we wish to determine whether $u \leq s$. However, (W-APP) also uses the subtyping jugdement in a second way: subtyping is a substitute for typing. Given a term t, we wish to find a value $v = (\lambda x \leq s.\ \mathsf{Top})$ such that v is a supertype of t.

This second use of the subtype relation is what inspired the idea of "subtyping as reduction." The supertypes of a term t can be obtained by evaluating t.

Unlike ordinary β-reduction, subtype reduction (i.e. $\xrightarrow{\leq}$) can promote variables to their bounding types. This means that a term can be reduced to a value even if that term contains free variables. For example, the term $x(y)$ is a normal form with respect to β-reduction; it cannot be further reduced. However, it can be reduced using subtype reduction, because x can be promoted to its bounding type. For example:

$$\begin{array}{l}\text{let } \Gamma = x \leq (\lambda z \leq \mathsf{Int}.\ z), y \leq \mathsf{Int} \\ \text{in } \Gamma \vdash_A\ x(y) \xrightarrow{\leq} (\lambda z \leq \mathsf{Int}.\ z)(y) \xrightarrow{\equiv} y \xrightarrow{\leq} \mathsf{Int}\end{array}$$

The $\xrightarrow{\leq}$ relation also differs from $\xrightarrow{\equiv}$ because it encodes an inequality. The choice of when and where promotions are performed will affect the final result, and even affect whether the result is well-formed or not. This section presents a concrete strategy for performing *subtype evaluation*: finding a reduction sequence $\Gamma \vdash\ t \xrightarrow{\leq}\!\!\!\!\rightarrow v$ that yields a value v.

The strategy we use has the following two characteristics. First, it yields a value v that is the *minimal* or *principal value supertype* of t. For all other values w, $t \leq w$ implies that $v \leq w$. Second, if t is well-formed, then the reduction sequence will only involve well-formed terms. This means that if $\Gamma \vdash_A\ t \xrightarrow{\leq}\!\!\!\!\rightarrow v$ is a minimal reduction sequence in the algorithmic system, then it is a valid well-subtyping judgement $\Gamma \vdash t \leq_{\text{wf}} v$ in the declarative system.

2.8.1 Subtyping is similar to abstract interpretation

In many ways, subtype reduction is similar to *abstract interpretation* [Cousot and Cousot, 1977] [Cousot, 2001]. Abstract interpretation works by replacing certain terms in a program with approximations of their values, and by redefining basic operations to handle approximate arguments. Evaluating a program with respect to approximate inputs will then yield an approximate answer.

A classic example of abstract interpretation is a system which replaces every in-

teger value with either "even" or "odd". Basic operations on integers are similarly redefined, e.g. "even + even = even", "even + odd = odd", etc. Evaluating an arithmetic expression within this framework will return an approximate answer that reflects whether the result is even or odd.

Subtype reduction works in a similar manner. A variable x represents a value that is not exactly known. However, we can perform computations by promoting x to its bounding type. The bounding type of x is an approximation which can be used in computations.

Note that subtyping is quite different from typing in this regard. In most traditional type systems, the type of a function $\lambda x : T.\ u$ is an arrow-type $T \to U$ or a Π-type $\Pi x : T.\ U$. Arrow-types and Π-types are constants; they are not functions. In contrast, the supertypes of $\lambda x \leq t.\ u$ are other functions, which can be used to perform computations.

2.8.2 Dealing with Top promotions

When choosing a reduction strategy, there are three basic reduction rules to consider: (1) β-reduction (SRE-APP), (2) variable promotion (SRS-VAR), and (3) Top promotion (SRS-TOP). The first two rules are syntax-directed. They can only be applied to specific terms (i.e. redexes and variables, respectively) so they work well with standard reduction strategies like left-most outer-most.

However, rule (SRS-TOP) is somewhat unusual. Any subterm in a covariant position can be promoted to Top, with no obvious way to choose when and where the rule should be applied. We resolve this dilemma by showing that all applications of (SRS-TOP) can be shifted to the end of a reduction sequence. In other words, a subtyping algorithm does not ever need to apply (SRS-TOP) until after all other reductions have been performed.

Lemma 2.8.1 (Reordering of a single Top-promotion)

Let $\Gamma \vdash_A t \xrightarrow{\leq}_{\mathsf{Top}} t'$ denote a subtype reduction that promotes some subterm of t to Top, using (SRS-TOP). If $\Gamma \vdash_A s \xrightarrow{\leq}_{\mathsf{Top}} t \xrightarrow{\leq} u$, then there exists a t', such that $\Gamma \vdash_A s \xrightarrow{\leq}{}^{=} t' \xrightarrow{\leq}_{\mathsf{Top}} u$, where $\xrightarrow{\leq}{}^{=}$ is the reflexive closure of $\xrightarrow{\leq}$.

Proof: By induction on $s \xrightarrow{\leq}_{\mathsf{Top}} t$. We note that Top promotions ignore the context Γ, and are preserved under substitution.

Case $s \xrightarrow{\leq} \mathsf{Top} \xrightarrow{\leq} \mathsf{Top}$: becomes $s \xrightarrow{\leq} \mathsf{Top}$.
Case $a(b) \xrightarrow{\leq}_{\mathsf{Top}} a'(b) \xrightarrow{\leq} a''(b)$: by induction.
Case $a(b) \xrightarrow{\leq}_{\mathsf{Top}} a'(b) \xrightarrow{\equiv} a'(b')$: becomes
$a(b) \xrightarrow{\equiv} a(b') \xrightarrow{\leq}_{\mathsf{Top}} a'(b')$.
Case $(\lambda x \leq a.\ b) \xrightarrow{\leq}_{\mathsf{Top}} (\lambda x \leq a.\ b') \xrightarrow{\leq} (\lambda x \leq a.\ b'')$. by induction.
Case $(\lambda x \leq a.\ b) \xrightarrow{\leq}_{\mathsf{Top}} (\lambda x \leq a.\ b') \xrightarrow{\equiv} (\lambda x \leq a'.\ b')$. becomes
$(\lambda x \leq a.\ b) \xrightarrow{\equiv} (\lambda x \leq a'.\ b) \xrightarrow{\leq}_{\mathsf{Top}} (\lambda x \leq a'.\ b')$.
Case $(\lambda x \leq a.\ b)(c) \xrightarrow{\leq}_{\mathsf{Top}} (\lambda x \leq a.\ b')(c) \xrightarrow{\equiv} [x \mapsto c]b'$ becomes
$(\lambda x \leq a.\ b)(c) \xrightarrow{\equiv} [x \mapsto c]b \xrightarrow{\leq}_{\mathsf{Top}} [x \mapsto c]b'$.
Case $(\lambda x \leq a.\ b)(c) \xrightarrow{\leq}_{\mathsf{Top}} \mathsf{Top}(c) \xrightarrow{\equiv} \mathsf{Top}$ becomes
$(\lambda x \leq a.\ b)(c) \xrightarrow{\leq}_{\mathsf{Top}} \mathsf{Top}$.
□

Lemma 2.8.2 (Reordering of a sequence of Top promotions) Any reduction sequence of the form $\Gamma \vdash_A t \xrightarrow{\leq}\!\!\!\!\rightarrow u$ can be reordered as $\Gamma \vdash t \xrightarrow{\leq}\!\!\!\!\rightarrow t' \xrightarrow{\leq}\!\!\!\!\rightarrow_{\mathsf{Top}} u$, such that $t \xrightarrow{\leq}\!\!\!\!\rightarrow t'$ contains no occurrences of (SRS-TOP).

Proof: By induction on the position of each occurence of (SRS-TOP) in the reduction sequence. Each application of lemma 2.8.1 will either eliminate a reduction, or move an occurence of (SRS-TOP) to the right. The number of reductions in the sequence does not increase. □

2.8.3 Minimal subtype reduction sequences

We now define *minimal reduction sequences*. Minimal reduction, written $\xrightarrow{\leq}_{\min}$, is defined as follows:

$$\frac{\Gamma \vdash_A t \xrightarrow{\equiv} t'}{\Gamma \vdash_A t \xrightarrow{\leq}_{\min} t'} \qquad \frac{\Gamma \text{ prevalid} \quad x \leq t \in \Gamma}{\Gamma \vdash_A x \xrightarrow{\leq}_{\min} t} \qquad \frac{\Gamma \vdash_A r^n \xrightarrow{\leq}_{\min} u}{\Gamma \vdash_A r^n(t^n) \xrightarrow{\leq}_{\min} u(t^n)}$$

where t^n, r^n are normal forms, defined syntactically as:

$t^n, u^n ::=$	$v^n \mid r^n$	normal forms
$v^n, w^n ::=$	$\mathsf{Top} \mid \lambda x \leq t^n.\ u^n$	value normal forms
$q^n, r^n ::=$	$x \mid r^n(t^n)$	residual (non-value) normal forms

Minimal reduction specifies a concrete evaluation strategy. Notice that variable promotion is only valid for residual normal forms. A minimal reduction sequence

$t \xrightarrow{\leq}\!\!\!\!\rightarrow_{\min} v^n$ must therefore reduce t to normal form using $\xrightarrow{\equiv}$ before applying any promotions. If the normal form is a value, then evaluation terminates at this point, since no other reductions apply. Otherwise the normal form must be a residual r^n. The variable on the spine of r^n is promoted (i.e the variable x in $x(t_1)...(t_k)$), and $\xrightarrow{\equiv}$ is used to generate a normal form again.

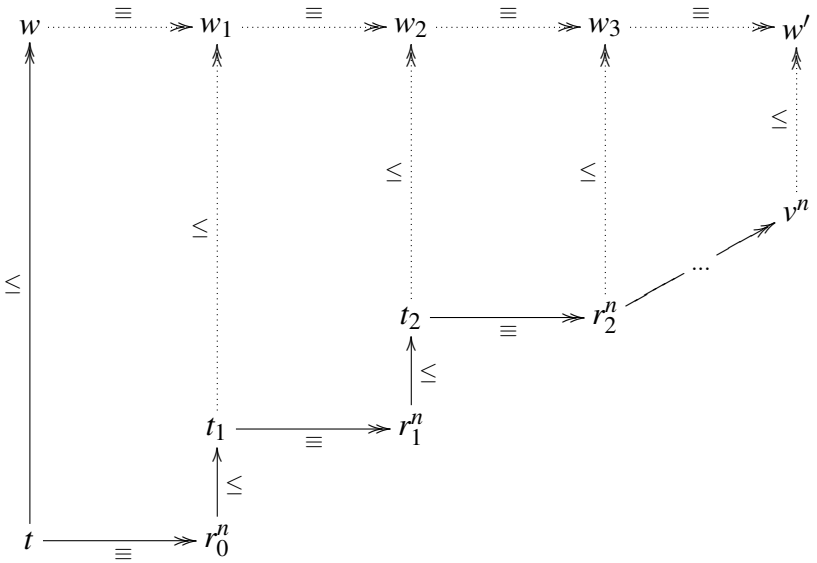
Since $\xrightarrow{\equiv}$ is confluent, the normal forms generated by this process must be unique. The reduction strategy used for $\xrightarrow{\equiv}$ doesn't matter; any valid reduction strategy can be used, such as call-by-name, call-by-value, or call-by-need.

Note that a minimal reduction sequence will never promote a term to Top, and it will never promote a variable inside a λ. We show that these two restrictions are sufficient to ensure that results are both minimal and well-formed.

Lemma 2.8.3 (If $t \xrightarrow{\leq}\!\!\!\!\rightarrow_{\min} v^n$, then v^n is the minimal supertype of t)

If $\Gamma \vdash_A t \xrightarrow{\leq}\!\!\!\!\rightarrow_{\min} v^n$, and $\Gamma \vdash_A t \xrightarrow{\leq}\!\!\!\!\rightarrow w$, then $\Gamma \vdash_A v^n \leq w$, assuming that $\xrightarrow{\equiv}$ and $\xrightarrow{\leq}$ commute.

Proof: If w is Top, then the result follows trivially. Otherwise, the proof is by by induction on the number of reduction steps that promote a variable in the sequence $t \xrightarrow{\leq}\!\!\!\!\rightarrow_{\min} v^n$. The proof is illustrated by the following diagram:



For the base case, there are no variable promotion steps; we have $t \xrightarrow{\equiv}\!\!\!\!\rightarrow v^n$. The result then follows by commutativity of $\xrightarrow{\leq}$ and $\xrightarrow{\equiv}$.

For the inductive case, we divide the reduction sequence $t \xrightarrow{\leq}\!\!\!\!\rightarrow_{\min} v^n$ into a series of segments, where each segment consists of a sequence of β-reductions, followed by a

variable promotion. In the first such segment, we have $t \xrightarrow{\leq}\!\!\!\!\rightarrow w$, and $t \xrightarrow{\equiv}\!\!\!\!\rightarrow r_0^n \xrightarrow{\leq}_{\min} t_1$. By commutativity, we obtain $w \xrightarrow{\equiv}\!\!\!\!\rightarrow w_1$, and $r_0^n \xrightarrow{\leq}\!\!\!\!\rightarrow w_1$.

The first step in the reduction sequence $r_0^n \xrightarrow{\leq}\!\!\!\!\rightarrow w_1$ must be a variable promotion for the following reason. The first step cannot be $\xrightarrow{\equiv}$, because r_0^n is a normal form. It must therefore be either a variable promotion, or a Top-promotion. By lemma 2.8.2, $r_0^n \xrightarrow{\leq}\!\!\!\!\rightarrow w_1$ can be reordered so that all promotions to Top occur at the end of the sequence. Thus, the first step will not be a Top-promotion unless the reordered sequence contains only Top-promotions. Since w is not Top, and r_0^n is not a value, the sequence $r_0^n \xrightarrow{\leq}\!\!\!\!\rightarrow w_1$ must have at least one reduction step that is not a Top-promotion.

We now know that $r_0^n \xrightarrow{\leq}_{\min} t_1$ promotes a variable and the first step in $r_0^n \xrightarrow{\leq}\!\!\!\!\rightarrow w_1$ also promotes a variable. There is only one place in any term where a variable can be promoted, so the first reduction step in $r_0^n \xrightarrow{\leq}\!\!\!\!\rightarrow w_1$ must be $r_0^n \xrightarrow{\leq} t_1$.

The remainder of the proof then follows by the induction hypothesis, using $t_1 \xrightarrow{\leq}\!\!\!\!\rightarrow w_1$, and $t_1 \xrightarrow{\leq}\!\!\!\!\rightarrow_{\min} v^n$. □

2.8.4 Well-formedness of minimal subtype reduction

We now show that minimal subtyping judgements preserve well-formedness: if t wf and $t \xrightarrow{\leq}_{\min} t'$, then t' wf. This property is important because it means that minimal reduction sequences can be used to construct well-subtyping judgements. Note that preservation does not hold for subtype reductions in general, as is illustrated the the following two counter-examples.

$$\begin{array}{lll} f \leq \mathsf{Top} \to \mathsf{Top}, b \leq \mathsf{Top} \vdash & f(b) & \xrightarrow{\leq} \quad \mathsf{Top}(b) \\ f \leq \mathsf{Top} \to \mathsf{Top}, b \leq \mathsf{Top} \vdash & (\lambda x \leq \mathsf{Top}.x)(f)(b) & \xrightarrow{\leq} \\ & (\lambda x \leq \mathsf{Top}.\mathsf{Top})(f)(b) & \xrightarrow{\leq} \quad \mathsf{Top}(b) \end{array}$$

In the first example, the function f is promoted to Top using rule (SRS-TOP). Whereas $f(b)$ is well-formed, $\mathsf{Top}(b)$ is not. The second example achieves a similar effect by promoting x to its bounding type, using rule (SRS-VAR). Thus, in general, neither variable promotion nor Top promotion is guaranteed to yield a well-formed reduct.

In both cases, the ill-formed term is a result of *over-generalization*. Every subtype reduction generalizes a term by discarding type information. In some cases, the discarded information is unnecessary, and the term remains well-formed without it. But in other cases, the type information is necessary, and discarding it yields a term that is

not well-formed.

Promoting a term to Top discards a great deal of information, and is often unsafe. Promoting a variable to its bounding type may also be unsafe if the variable occurs inside a redex. Promoting inside a redex and then contracting the redex will always yield a more general type than simply contracting the redex immediately. (This fact can be seen in the commutivity diagrams for $\xrightarrow{\equiv}$ and $\xrightarrow{\leq}$.)

Minimal subtype reductions avoid both of these cases. A minimal reduction never promotes to Top, and it will never step inside a λ-abstraction. In essence, a minimal reduction is guaranteed to yield a well-formed result precisely because it is minimal; it avoids over-generalizing by discarding the minimum possible amount of information.

Lemma 2.8.4 (Minimal reductions preserve well-formedness)
If $\Gamma \vdash t$ wf, and $\Gamma \vdash_A t \xrightarrow{\leq}_{\min} t'$, then $\Gamma \vdash t'$ wf.

Proof: By induction on t wf.

Case $t \xrightarrow{\equiv} t'$. By theorem 2.5.12 (type safety).

Case $x \xrightarrow{\leq}_{\min} t'$. By lemma 2.5.1 (context formation).
If x wf then Γ wf, and if $\Gamma_0, x \leq t', \Gamma_1$ wf then $\Gamma_0 \vdash t'$ wf.

Case $r^n(t^n) \xrightarrow{\leq}_{\min} u(t^n)$. Since $r^n(t^n)$ wf, we have $r^n \leq_{\text{wf}} (\lambda x \leq a.\ \text{Top})$. By theorem 2.6.20, this judgment is valid in the algorithmic system, and thus can be interpreted as a sequence of reductions. Since r^n is a normal form, the first reduction in $r^n \leq \lambda x \leq a.\ \text{Top}$ must be $r^n \xrightarrow{\leq}_{\min} u$; the logic is the same as that used in the proof of lemma 2.8.3. Translating back to the declarative system, we have $\Gamma \vdash u \leq (\lambda x \leq a.\ \text{Top})$, and $u(t^n)$ wf.

□

2.8.5 Minimal subtype comparisons

At the beginning of this section, we identified a need for two different subtyping algorithms. The first algorithm involves finding the principal value supertype of a term. Given a term t, the principle supertype is the minimal value v^n, such that $t \leq v^n$. We now turn our attention to the second, and more more common use of subtyping: comparison. Given two terms t and u, we define a practical algorithm for determining whether $t \leq u$.

The following rules define a practical algorithm for deciding whether $t \leq u$. We

refer to this algorithm as *canonical subtyping*.

$$\frac{\Gamma \vdash_A t \xrightarrow{\equiv}\!\!\!\!\rightarrow t^n, \quad u \xrightarrow{\equiv}\!\!\!\!\rightarrow u^n, \quad t^n \leq_{\text{cnl}} u^n}{\Gamma \vdash_A t \leq_{\text{cnl}} u} \quad \text{(CS-RED)}$$

$$\frac{\Gamma \vdash_A r^n \xrightarrow{\leq}_{\min} t, \quad t \leq_{\text{cnl}} u^n}{\Gamma \vdash_A r^n \leq_{\text{cnl}} u^n} \quad \text{(CS-PROM)}$$

$$\Gamma \vdash_A r^n \leq_{\text{cnl}} r^n \quad \text{(CS-REFL)}$$

$$\Gamma \vdash_A t^n \leq_{\text{cnl}} \mathsf{Top} \quad \text{(CS-TOP)}$$

$$\frac{\Gamma, x \leq t^n \vdash_A u^n \leq_{\text{cnl}} s^n}{\Gamma \vdash_A (\lambda x \leq t^n.\ u^n) \leq_{\text{cnl}} (\lambda x \leq t^n.\ s^n)} \quad \text{(CS-FUN)}$$

Canonical subtyping is very similar to the definitions of algorithmic subtyping for System $F^{\omega}_{\leq}$ found in [Steffen and Pierce, 1994] and [Compagnoni and Goguen, 2003]. In essence, the subtype relation is defined only over normal forms; any terms which are not in normal form must be reduced to normal form before proceeding.

A normal form is either a residual r^n or a value. (See Section 2.8.3 for the definition of residuals.) A comparison between residuals is by simple syntactic equality (CS-REFL), while a comparison between functions proceeds by comparing their bodies (CS-FUN). If neither of these rules is applicable, then the residual on the left can be promoted using minimal subtype reduction (CS-PROM).

2.8.5.1 Canonical subtyping is transitive

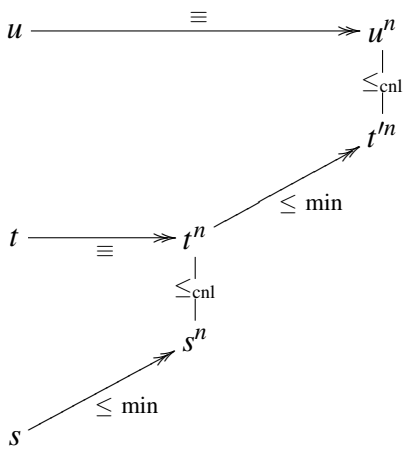
One major advantage of canonical subtyping is that transitivity is admissible, without having to resort to commutativity of $\xrightarrow{\equiv}$ and $\xrightarrow{\leq}$.

Lemma 2.8.5 (Canonical subtyping is transitive)

If $\Gamma \vdash_A s \leq_{\text{cnl}} t, \quad t \leq_{\text{cnl}} u,$ then $\Gamma \vdash_A s \leq_{\text{cnl}} u.$

Proof: By induction on $s \leq_{\text{cnl}} t$.
Case ((CS-REFL),(CS-REFL)): trivial, using (CS-REFL).
Case ((CS-FUN), (CS-FUN)): By the induction hypothesis, using (CS-FUN).
Case (any, (CS-TOP)): trivial, using (CS-TOP).
Case ((CS-TOP), any): trivial, using (CS-TOP).

In all other cases, we observe that a derivation of $\Gamma \vdash_A s \leq_{\text{cnl}} t \leq_{\text{cnl}} u$ must reduce s, t, and u according to the following diagram:



If t^n is a residual, then $s^n = t^n$, and therefore $s^n \xrightarrow{\leq}_{\min} t'^n$, which proves the result. If t^n is a λ-abstraction, then then $t^n = t'^n$, and the result follows by the inductive hypothesis on $s^n \leq_{\text{cnl}} t^n \leq_{\text{cnl}} u^n$. $\square$

2.8.6 Decidability and completeness

The last question we wish to address is whether the two algorithms given in this section (finding principle supertypes, and canonical subtyping) are decidable and complete. The answer is no in both cases.

It is clear that both algorithms are undecidable, because they both involve reducing terms to normal form. They are thus decidable only if reduction is guaranteed to terminate, but as discussed in Section 2.4.4, System $\lambda_{\triangleleft}$ is not strongly normalizing.

Both algorithms are incomplete for a similar reason. In the case of principle supertypes, a complete algorithm would, given a term t, return a value v such that $t \leq v$ if such a value exists. Note that an undecidable algorithm can still be complete if it fails to terminate only in the case where v does not exist.

We show that the both algorithms are incomplete by giving a counter-example. According to Girard's paradox, the type $(\Pi x : *.\, x)$ is inhabited in System $\lambda*$. Let ω be the term that has this type. Given any type T, $\omega(T)$ is a non-terminating term of type T. Using the embedding from System $\lambda*$ into System $\lambda_{\triangleleft}$, we have $\omega(T) \leq T$ for any term T. We set $T = (\lambda x \leq \mathsf{Top}.\, \mathsf{Top})$, which is a value. $\omega(T)$ thus has a supertype, but the minimal subtyping algorithm will fail to find it, because the evaluation of $\omega(T)$ will not terminate. The canonical subtype algorithm will similarly fail to terminate when

comparing $\omega(T) \leq T$, even though that judgement is derivable using typing rules from System $\lambda*$.

2.8.6.1 Completeness and transitivity in System $F^{\omega}_{\leq}$

The canonical subtype algorithm given here is essentially the same as the one defined for System $F^{\omega}_{\leq}$ in [Steffen and Pierce, 1994] and [Compagnoni and Goguen, 2003]. In System $F^{\omega}_{\leq}$, the algorithm is shown to be both decidable and complete. It is decidable because the language of types in System $F^{\omega}_{\leq}$ is strongly normalizing, and it is complete because transitivity is admissible.

The failure of strong normalization in System $\lambda_{\lhd}$ means that we have been forced to take a very different approach. We show that transitivity elimination and type soundness follow from a confluence property: the commutativity of $\xrightarrow{\equiv}$ and $\xrightarrow{\leq}$. However, a proof of commutativity would not be necessary if the language was strongly normalizing.

2.8.7 Usability

The fact that the principle supertype and canonical subtyping algorithms are undecidable and incomplete does not mean that they are unusable in practice. The call-by-value reduction strategy [Plotkin, 1975] is also undecidable and incomplete for essentially the same reason. The goal of evaluation is to reduce a term t to a value v in weak head normal form. Call-by-name and call-by-need evaluation are undecidable but complete; they will find a v if one exists. Call-by-value is not complete; it may fail to terminate even if a v exists. Nevertheless, call-by-value is the most common strategy used to implement programming languages, because it is simple to implement, and has good space and time complexity in the common case.

We have implemented both the principal value supertype algorithm, and the canonical subtyping algorithm in the current implementation of the DEEP calculus, which is described in Chapter 4.

2.9 Past and future work

Thus far, we have shown that system $\lambda_{\lhd}$ is type safe if it has a property known as transitivity elimination. In an effort to prove this property, we reformulated the subtype relation as an abstract reduction system, and showed that transitivity elimination

follows from commutativity of the underlying reductions. We were able to establish that $\xrightarrow{\leq}$ and $\xrightarrow{\equiv}$ commute locally, but we were unable to establish that they commute globally. Our proof of type safety is therefore incomplete.

We believe that our proof technique is a good one, and it has certain advantages over previous techniques that have been proposed in the literature. However, it is not the only technique that could potentially be applied. Since our proof is incomplete, it is important to examine other possibilities.

This section provides a brief summary of the problem, describes how previous work in the literature has dealt with it, and compares previous work with our approach. We then turn our attention to the future, and discuss a number of alternative ideas that are promising avenues for future research.

2.9.1 Transitivity elimination: a summary of the problem

Subtyping is inherently harder to deal with than typing because subtyping is transitive, whereas typing is not. Because the typing relation lacks a transitivity rule, typing is *invertible*: the typing derivation for a term is uniquely determined by the syntax of that term. The inversion lemma (sometimes called the generation lemma) is a crucial part of most type safety proofs.

As explained in Sections 2.5.3 and 2.5.4, the presence of a transitivity rule prevents us from proving an inversion lemma for subtyping. We need to show that $(\lambda x \leq a.\ b) \leq (\lambda x \leq a'.\ b)$ if **and only if** $a \equiv a'$.

At the same time, we need the transitivity rule in order to prove two other crucial lemmas: substitution and narrowing. For example, given the redex $(\lambda x \leq a.\ b)(c)$, we must show that it is safe to substitute c for x. In general, this proof requires transitivity; every subderivation of the form $x \xrightarrow{\leq} a \leq a'$ becomes $c \leq a \leq a'$. Note that although we have written the promotion of x to a as a reduction, a similar promotion rule can be found in the definition of algorithmic subtyping for any theory that supports bounded quantification [Pierce, 2002] [Steffen and Pierce, 1994] [Compagnoni, 1995] [Chen, 1999] [Zwanenburg, 1999] [Compagnoni and Goguen, 2003].

Standard practice in this situation is to formulate an algorithmic subtyping judgement that does not have a transitivity rule, and which therefore does have an inversion lemma. Transitivity is then shown to be admissible within the algorithmic system, which allows the proofs of substitution and narrowing to go through. This is also the approach that we have taken for System $\lambda_{\lhd}$.

2.9.2 A brief history of higher-order subtyping

Transitivity elimination for systems with higher-order subtyping is known to be a hard problem. The first version of System $F^{\omega}_{\leq}$, developed by Steffen and Pierce, has an unusual restriction. Although it allows bounded quantification on functions at the object level, it still uses kinding for type operators. In other words, a polymorphic function is written as $\lambda X \leq T.\ u$, where T is a type, and u is an object. A type operator, however, is written as $\lambda X : K.\ U$, rather than $\lambda X \leq T.\ U$. This decision breaks the symmetry of the language, and decreases its expressiveness.

The reason Steffen and Pierce made the restriction is because it simplifies the metatheory. As they write in a footnote: "The more general form of this property... would be much more difficult to prove" [Steffen and Pierce, 1994]. Compagnoni makes the same decision in her work [Compagnoni, 1995], and Chen does the same thing when adding subtyping to the calculus of constructions [Chen, 1997] [Chen, 1999] [Castagna and Chen, 2001].

Zwanenburg's formulation of subtyping for Pure Type Systems includes both bounded quantification and kinded quantification [Zwanenburg, 1999]. Although this seems like an unnecessary duplication of syntax, there is a subtle reason for including both forms. Zwanenburg only allows higher-order subtyping (i.e. point-wise subtyping) on the kinded operators; operators with bounded quantification only have trivial subtypes.

Both Steffen and Pierce's restriction, and Zwanenburg's restriction prevent bounded quantification from being combined with higher-order subtyping. If this combination is prohibited, then the following confluence diagram does not arise, because x cannot be promoted to a. As discussed in Section 2.7.4, this particular case is what causes the diamond property of $\xrightarrow{\leq}$ and $\xrightarrow{\equiv}$ to fail. If the diamond property did not fail, then we would have a proof that $\xrightarrow{\leq}$ and $\xrightarrow{\equiv}$ commute, that transitivity is admissible, and that System $\lambda_{\lhd}$ is sound.

$$\begin{array}{ccc} (\lambda x \leq a.\ C_{\leq}[a])(c) & \overset{\equiv}{\dashrightarrow} & C_{\leq}[a] \overset{\equiv}{\dashrightarrow} C_{\leq}[d] \\ \uparrow \leq & & \uparrow \leq \\ (\lambda x \leq a.\ C_{\leq}[x])(c) & \xrightarrow[\equiv]{} & C_{\leq}[c] \end{array} \qquad \text{where } c \overset{\leq}{\dashrightarrow} d \overset{\equiv}{\dashleftarrow} a$$

To our knowledge, the only type theory in the literature which successfully combines higher order subtyping and bounded quantification is the version of System $F^{\omega}_{\leq}$

presented by Compagnoni and Goguen [Compagnoni and Goguen, 2003]. They use a proof technique called "typed operational semantics", in which every judgment comes equipped with a proof that the terms in question have normal forms. This technique can only be applied to languages that are strongly normalizing.

The proof techniques used in this chapter demonstrate why strong normalization is useful. In our proof, we are able to show that subtype reductions commutate locally, but we have been unable to show that subtype reductions commutate globally. According to Newman's lemma, however, any system which is both weakly confluent (or weakly commutive) and strongly normalizing is confluent (or commutative) [Bezem et al., 2003].

By formulating the subtype relation as an abstract reduction system, we have a developed a general framework in which the results of other theories can be compared. Every other theory of subtyping in the literature has a restriction that, if applied to System $\lambda_{\lhd}$, would cause our proof technique to succeed as well.

2.9.3 Commutativity by strong normalization of bounding types

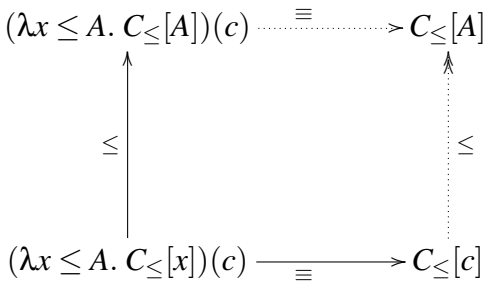
The most straightforward way to complete the proof of transitivity elimination for System $\lambda_{\lhd}$ would be strengthen the system so that it is strongly normalizing. As it turns out, it is not necessary for all terms to be strongly normalizing, it is only necessary for terms that are used as bounding types.

For example, consider the following subset of terms, which we shall call pseduo-types, denoted by the meta-variable A:

$$A \quad ::= \quad \mathsf{Top} \mid \lambda x \leq A.\, A$$

The pseudo-types correspond roughly to the kinds of System F_ω. However, note that we have not defined a separate notion of type or kind; every pseudo-type is a legal term in System $\lambda_{\lhd}$, and can thus act as a bounding type for other terms, using the subtype relation that we have already defined. The psuedo-types are clearly strongly normalizing, because there are no redexes, and they are preserved under substitution, because there are no variables.

Now consider what happens if we restrict the syntax of System $\lambda_{\lhd}$ so that functions have the form $\lambda x \leq A.\, u$, with a psuedo-type as the argument bound. The resulting system has the transitivity-elimination property, because the following diagram is now commutes strongly, rather than locally:



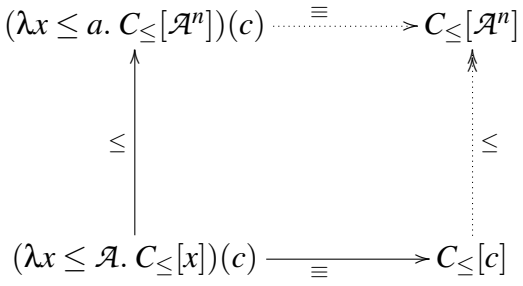
where $c \xrightarrow{\leq}\!\!\!\!\rightarrow A$

Because the pseudo-type A is already in normal form, it cannot be further reduced on the top edge of the diagram. Multiple reductions on the right side of the diagram are not a problem, because we can use the number of reductions on the top and bottom edges as our inductive hypothesis.

This same technique can be extended to more sophisticated definitions of psuedo-types. Let $\mathcal{A}$ be a set of terms in System $\lambda_{\lhd}$ that are strongly normalizing, and stable under substitution. As before, we restrict the calculus so that functions have the syntax $\lambda x \leq \mathcal{A}.\ u$. We now modify (SRS-PROM), the variable promotion rule, to the following:

$$\frac{\Gamma \text{ prevalid} \quad x \leq \mathcal{A} \in \Gamma \quad \mathcal{A} \xrightarrow{\equiv}\!\!\!\!\rightarrow \mathcal{A}^n}{\Gamma \vdash\ x \xrightarrow{\leq} \mathcal{A}^n} \text{ where } \mathcal{A}^n \text{ is in normal form.}$$

This change will cause the confluence diagram to become:



where $c \leq \mathcal{A} \xrightarrow{\equiv}\!\!\!\!\rightarrow \mathcal{A}^n$

Because equivalence reduction ($\xrightarrow{\equiv}$) is confluent, we can derive $c \xrightarrow{\leq}\!\!\!\!\rightarrow \mathcal{A}^n$ using $c \leq \mathcal{A}$ (from the premise of rule (SRE-APP)), and $\mathcal{A} \xrightarrow{\equiv}\!\!\!\!\rightarrow \mathcal{A}^n$, which we get from the new definition of (SRS-PROM).

In a sense, it is not suprising that we can eliminate transitivity by defining a set of pseudo-types that are strongly normalizing. The vast majority of type systems are structured in this way. The language of objects may include general recursion or other features that break strong normalization, so long the language of types does not have these features.

2.9.3.1 Universes and impredicativity

The difficulty in applying this technique lies in the details: how do we define a set of psuedo-types that are expressive enough to be useful, yet still strongly normalizing? The failure of strong normalization is due to the presence of a fully impredicative Top, which gives rise to Girard's paradox. Eliminating Top would give us a predicative system. Although we have not proven strong normalization for a predicative version of System $\lambda_{\lhd}$, we suspect that the usual proof techniques (i.e. logical relations) would apply.

Completely eliminating Top is not necessarily the best approach, however, because Top is useful; it is a necessary part of our embeddings of System $F_{\leq}$, $F^{\omega}_{\leq}$, and pure type systems. Nevertheless, as discussed in Section 2.3.4, it is not hard to add a universe structure to System $\lambda_{\lhd}$. The simple universe structure given earlier is still fully impredicative, and thus admits Girard's paradox. However, a more sophisticated universe structure such as that found in the extended calculus of constructions [Luo, 1994] eliminates paradox, while still allowing some impredicativity, and it is possible that such a structure could be translated to System $\lambda_{\lhd}$.

We have not explored this area further because any fix based on universes would not scale to the DEEP calculus, which is the subject of the next two chapters. The DEEP calculus includes general recursion, which makes it far easier to construct nonterminating expressions than Girard's paradox does. Moreover, recursion in DEEP is not restricted to the object level, because DEEP supports recursive types. One of the advantages of using pure subtype systems is that type-level recursion and object-level recursion in DEEP have the same semantics; this advantage is crucial to our handling of modules, and would be lost if we attempted to use universes to enforce strong normalization of types.

2.9.4 A simpler conditional rewrite system

There are two main benefits to formulating the subtype relation as an abstract reduction system (ARS). The primary benefit is that our proof technique is based on commutativity of reductions, and commutativity does not necessarily require strong normalization. Since the DEEP calculus includes general recursion over types, this is a key advantage. The other main benefit is that there is an extensive literature on ARSs which could potentially be applied.

Unfortunately, subtype reduction in System $\lambda_{\lhd}$ has a number of features that make

it difficult to study. First, subtype reduction is an example of a *higher-order rewrite system* [Klop et al., 1993], since it contains higher-order functions and bound variables. Second there are two different reductions: $\xrightarrow{\leq}$ and $\xrightarrow{\equiv}$, and the two reductions are not orthogonal. Third, reductions must be done within a context Γ. And finally, infinite reduction sequences exist but are hard to construct.

Most of the general theory of ARSs has been built around models that do not have these properties. To simplify matters, we have thus devised the following reduction system, which demonstrates confluence behavior that is very similar to subtype reduction in System $\lambda_{\lhd}$, but is formulated as a conventional ARS.

Terminals: A Non-terminals: C, D

Let $=$ be the symmetric and transitive closure of $\longrightarrow$.

$$
\begin{array}{lcll}
A & \longrightarrow & C(A) & \\
D(x,y) & \longrightarrow & x & \text{if } x = y \\
D(x,y) & \longrightarrow & y & \text{if } x = y
\end{array}
$$

The above system is a standard first-order conditional rewrite system. There is only one kind of reduction, and no contexts. Like System $\lambda_{\lhd}$, it is not strongly normalizing, because the $A \longrightarrow C(A)$ rule generates an infinite reduction sequence. The conditions are equality conditions rather than join conditions, so there is no obvious counter-example to confluence (see Section 2.6.3). There is also no obvious proof of confluence, because the rules are not *orthogonal* [Bezem et al., 2003]; a $D(x,y)$ term can be reduced in two different ways.

Much like subtype reduction, the above rewrite system gives rise to the following diagram of local confluence:

$$
\begin{array}{ccc}
D(a,b) & \longrightarrow & a \\
\downarrow & & \downarrow \\
b & \longrightarrow\!\!\!\!\!\rightarrow & \cdot
\end{array}
$$

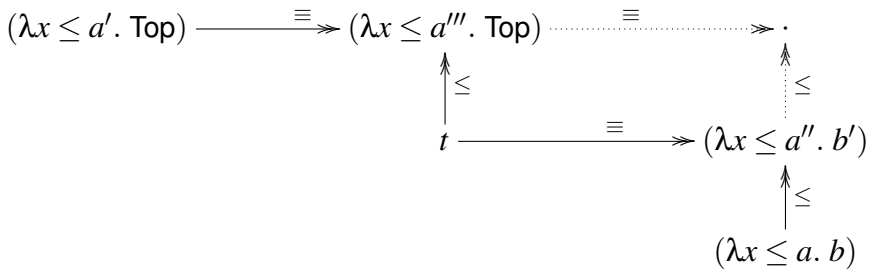
The condition on $D(a,b) \longrightarrow a$ tells us that $a = b$. Given an appropriate induction hypothesis, we could transform $a = b$ into a transitivity-free form, giving us $a \longrightarrow\!\!\!\!\!\rightarrow \cdot \longleftarrow\!\!\!\!\!\leftarrow b$, which are the completing edges of the diagram. Notice that we complete the diagram by using the condition on one of the rewrite rules; this way of completing diagrams is identical to what we did for subtype reduction in Section 2.7.4.

We conjecture that if a proof of confluence can be derived for the above rewrite system, then that proof can be adapted to show that $\xrightarrow{\leq}$ and $\xrightarrow{\equiv}$ commute in System

$\lambda_{\lhd}$. Moreover, if a counter-example to confluence can be derived for the above rewrite system, then that counter-example can also be adapted to disprove commutativity, and consequently type safety, for System $\lambda_{\lhd}$.

2.9.5 Partial transitivity elimination

The standard approach to dealing with subtyping in the literature, and the approach that we have pursued in this chapter, is to show full transitivity elimination. However, for the purpose of proving type safety, transitivity does not have to be completely eliminated. It only needs to be eliminated when comparing values, e.g.



This is a diagram of single-step transitivity elimination for the derivation $(\lambda x \leq a.\ b) \leq (\lambda x \leq a'.\ \mathsf{Top})$. Notice that in this particular case, we know that the intermediate term t reduces to a value. Since call-by-name reduction [Plotkin, 1975] is guaranteed to yield a value if one exists, the number of call-by-name reductions to a value might be used to strengthen the induction hypothesis.

Strengthening the induction hypothesis in this way would not give us a complete proof of transitivity elimination, but would demonstrate that transitivity could be eliminated for the particular cases that are necessary for type safety. Unfortunately, there are significant challenges to applying this observation to the rest of the proof. We have not addressed multi-step transitivity elimination, or explained how the stronger induction hypothesis could be lifted to subterms, both of which are necessary steps.

Along similar lines, it might also be possible to abandon our definition of subtyping as an abstract reduction system, and switch to a more conventional proof in which occurences of transitivity are pushed deeper into the branches of proof tree. It is not necessary to eliminate transitivity entirely; occurences of transitivity only need to be pushed "deep enough" so that a derivation which compares two functions is guaranteed to end in rule (DS-FUN).

2.9.6 Circular dependencies and typed operational semantics

One of the difficulties in working with System $\lambda_{\lhd}$ is that the system has a number of circular dependencies that are difficult to unravel. Such dependencies are not unique to System $\lambda_{\lhd}$; they are common in other theories of subtyping as well; Section 5.3.2 provides an overview of the literature.

In System $\lambda_{\lhd}$, well-formedness depends upon subtyping, but subtyping is only defined over well-formed terms. We attempted to break this dependency in the definition of algorithmic subtyping by defining subtype reduction over all terms. However, this merely uncovered another circular dependency: the substitution lemma requires transitivity, but the diagrams for confluence and commutativity, which are required for transitivity, involve substitution. We resolved this dependency by using $\lhd^*$ (the transitive closure of $\lhd$), rather than $\lhd$ itself in the premise of rule (SRE-APP) (see Figure 2.7). This change also resolved another circular dependency related to conditional rewrite systems; see section 2.6.3.

Typed operational semantics [Goguen, 1995] is a technique that handles such dependencies in a different way. In traditional type theory, there is a close relationship between typing and evaluation. Rather than splitting the two judgements apart, typed operational semantics fuses them together. Every typing derivation comes equipped with a proof that terms have normal forms, and similarly, every evaluation derivation is equipped with a proof that terms are well-typed. Fusing the two judgements results in significantly simpler proofs of certain properties. In System $F^{\omega}_{\leq}$, which is closely related to System $\lambda_{\lhd}$, typed operational semantics is used to handle the circular dependency between transitivity elimination, strong normalization, and type soundness [Compagnoni and Goguen, 2003].

Typed operational semantics itself is not appropriate for System $\lambda_{\lhd}$ because System $\lambda_{\lhd}$ is not strongly normalizing. However, the basic idea could potentially be adapted to prove other properties, such as commutativity. In essence, the technique works by adding a number of additional premises to the inference rules for typing. The presence of these premises strengthens the induction hypothesis, and a stronger induction hypothesis is exactly what we need for System $\lambda_{\lhd}$.

2.9.7 Logical relations

The technique of logical relations is another powerful proof technique that was initially invented to prove strong normalization, but can be used to prove other properties

as well [Tait, 1975] [Girard et al., 1989] [Gallier, 1990]. Strong normalization is difficult to prove because it is not possible to use simple induction on the size of terms; β-reduction changes both the size and shape of terms. A proof by logical relations proceeds instead by induction on the size of types. Each type denotes a set of terms which have a particular property (such as normalizability), and larger types (e.g. $T \to U$) denote sets of terms that preserve the properties of smaller types (e.g. T).

In systems with higher-order subtyping, the proof of transitivity elimination is hard for the same reason that strong normalization proofs are hard: higher-order subtyping involves β-reduction, which changes the size and shape of terms. At first glance, the logical relations technique thus appears like a promising way of approaching the problem. Moreover, in a first-order type theory, extending the technique to subtyping is not difficult, since subtypes merely denote subsets.

The difficulty in applying the logical relations technique to System $\lambda_{\lhd}$ is twofold. The first and most obvious problem is that there are no types per se, so it is not possible to perform induction on the size of the type. "Types" in System $\lambda_{\lhd}$ are simply ordinary terms. One might attempt to use induction on the size of normal forms, but normal forms do not necessarily exist.

A second and more subtle problem is that Top in System $\lambda_{\lhd}$ is fully impredicative, and quantifies over all terms. This introduces a potential circularity in the proof; it becomes necessary to assume that all terms have a certain property, in order to prove that a subset of them have that property, which is clearly nonsense. Girard's "Candidates of Reducibility" [Girard et al., 1989] [Gallier, 1990] work for the limited amounts of impredicativity of System F, but not for the full impredicativity of System $\lambda_{\lhd}$.

In order to use the logical relations technique, we would need to either eliminate Top, or impose a universe structure of some kind. Once again, this technique would not necessarily scale to DEEP, because DEEP includes recursive types, which also make the logical relations technique difficult to apply.

2.9.8 Contravariance

Contravariance is a common extension to many theories of subtyping [Pierce, 2002]. Contravariance extends the rule for functions (or arrow-types) so that the argument bound is allowed to change in subtypes:

$$\frac{\Gamma \vdash\ t' \leq t \quad \Gamma, x \leq t' \vdash\ u \leq u'}{\Gamma \vdash\ \lambda x \leq t.\ u\ \leq\ \lambda x \leq t'.\ u'}$$

System $\lambda_{\lhd}$ does not currently support contravariance. Rule (DS-FUN) has the restriction that $t' \equiv t$, rather than $t' \leq t$. We have chosen not to support contravariance because it is not required for the problems we wish to solve, and because contravariance has several well-known pathologies [Pierce, 1994] [Pierce, 2002].

Nevertheless, it is important to consider whether our approach to algorithmic subtyping could support contravariance if need be. In other words, is it possible to formulate contravariant subtyping as a reduction system? As it turns out, it is possible to do so, but the resulting system does not have the transitivity elimination property.

2.9.8.1 Contravariance breaks transitivity

In order to support contravariance, we would need to add a third kind of reduction. The three kinds are:

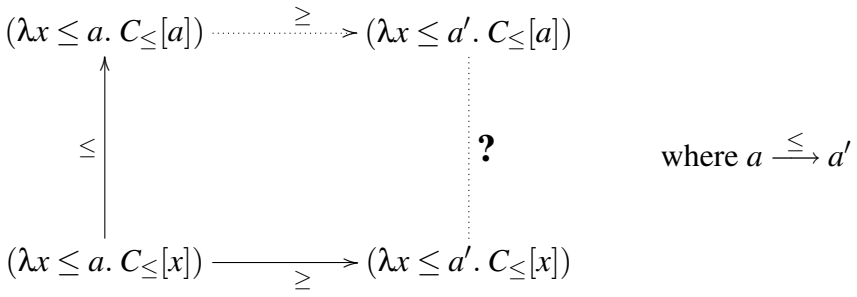
- $t \xrightarrow{\leq} t'$ rewrites a term t to a supertype t'.
- $t \xrightarrow{\equiv} t'$ rewrites a term t to an equivalent term t'.
- $t \xrightarrow{\geq} t'$ rewrites a term t to a subtype t'. (contravariant reduction)

Recall that $\xrightarrow{\leq}$ generates a supertype by promoting variables to their bounding types, or by promoting subterms to Top, and that such promotions can only happen in a covariant position. The $\xrightarrow{\geq}$ reduction does the exact same thing, but it does so in a contravariant position, as illustrated below.

$$\lambda x \leq t.\, x \xrightarrow{\leq} \lambda x \leq t.\, \mathsf{Top} \quad \text{(covariant reduction)}$$
$$\lambda x \leq t.\, x \xrightarrow{\geq} \lambda x \leq \mathsf{Top}.\, x \quad \text{(contravariant reduction)}$$

The overall definition of subtyping must be altered as well, to take the new form of reduction into account. In System $\lambda_{\lhd}$, $t \leq u$ if and only if $t \xrightarrow{\leq}\!\!\!\!\rightarrow \cdot \;\!\!\!\!\leftarrow\!\!\!\!\!\xleftarrow{\equiv} u$. With contravariance, the definition should be $t \leq u$ iff $t \xrightarrow{\leq}\!\!\!\!\rightarrow \cdot \;\!\!\!\!\leftarrow\!\!\!\!\!\xleftarrow{\leq} u$. (Here $u' \xleftarrow{\leq} u$ is the same as $u \xrightarrow{\geq} u'$).

This change introduces additional commuting diagrams to the proof of transitivity elimination. Instead of showing that $\xrightarrow{\leq}$ and $\xrightarrow{\equiv}$ commute, we would need to show that $\xrightarrow{\leq}$ and $\xrightarrow{\geq}$ commute. As it turns out, these two reductions do *not* commute, as shown in the following diagram. A promotion from x to a on the left edge cannot be matched on the right edge, because the bounding type of x has changed from a to a'.



Another way of interpreting this result is to observe that the algorithmic definition of contravariant subtyping that we give here is significantly different from the original declarative rule given at the start of this section. The algorithmic definition only allows us to derive the following:

$$\frac{\Gamma \vdash\ t' \xrightarrow{\leq}\!\!\!\!\rightarrow t'' \xleftarrow{\leq} t \qquad \Gamma, x \leq t'' \vdash\ u \leq u'}{\Gamma \vdash\ \lambda x \leq t.\ u\ \leq\ \lambda x \leq t'.\ u'}$$

Notice that the context is $\Gamma, x \leq t''$ in the algorithmic version, whereas it is $\Gamma, x \leq t'$ in the declarative version. The algorithmic definition closely resembles the following rule for System $F_{\leq}$:

$$\frac{\Gamma \vdash\ T' \leq T \qquad \Gamma, X \leq T \vdash\ U \leq U'}{\Gamma \vdash\ \Pi X \leq T.\ U\ \leq\ \Pi X \leq T'.\ U'}$$

The above rule is sometimes called the S-ALL-LOC rule. It was originally proposed as an alternative to the usual contravariant subtyping rule (which has $\Gamma, X \leq T'$), since the usual rule causes subtyping in System $F_{\leq}$ to be undecidable. The S-ALL-LOC rule was abandoned after Giorgio Ghelli demonstrated that it broke transitivity elimination, using a proof very similar to the commuting diagram shown above [Ghelli, February 24, 1993].

2.9.9 Denotational semantics

Denotational semantics [Tennent, 1976] is very different from the syntactic approaches that we have considered thus far. Denotational semantics assigns a *denotation* to every well-typed (or well-formed) term. The denotation of a term is a mathematical structure that captures the "meaning" of the term in some sense, e.g. λ abstractions correspond to mathematical functions.

Denotational semantics has an important property, namely that all equivalent terms have the same denotation. Equivalence in this case includes at least reduction, but

may include other measures of observational equivalence as well. For example, the denotation $\perp$ is commonly given to all terms which do not have a normal form, regardless of whether they are syntactically equivalent or not. Because of this structure, it is possible to study properties of denotations without considering syntactic issues, like β-convertibility, which complicate the operational semantics.

The structure of System $\lambda_{\lhd}$ seems well-suited to denotational models. The subtype relation over terms would give rise to a partial order over denotations. Moreover, the ordering is easy to define: if we assume that denotations are functions, then we have $f \leq g$ iff $f(a) \leq g(a)$ for all a. Because of the partial order, every term t would have both a denotation and a domain. The domain of t would be the set of all denotations that are less than the denotation of t according to the subtype ordering. This observation reflects the fact that every term can behave as both a type (i.e. a domain) and an object (i.e. the denotation of the term).

Once again, the devil is in the details. What, precisely, should the denotations of terms in System $\lambda_{\lhd}$ be? Is it possible to construct a compositional model? Is the denotation of subtyping well-defined, or does impredicativity introduce circularities? This is a promising avenue for future research, but it is not one that we have explored in any detail to date.

2.10 Discussion

Type systems with subtyping are very expressive. As earlier work on System $F^{\omega}_{\leq}$ shows, bounded quantification is powerful enough to completely replace kinding, because every kind can be easily encoded as a Top-type (see section 2.2.2). System $\lambda_{\lhd}$ extends this result to typing in general, as shown in Section 2.4. As is always the case in type theory, however, such expressiveness comes at a cost, and in this case, the cost is the complexity of the meta-theory.

This thesis raises a very important question, and one which is as yet unanswered. *Is it meaningful to talk about a type expression that does not terminate?* The study of type theory in the literature has overwhelmingly focused on systems that are strongly normalizing, and for good reason. One reason is practical: compiler writers want to make sure that the compiler does not hang. Another reason is theoretical: it is much easier to prove important properties, such as soundness and completeness, for type theories in which typing is decidable. The final reason, and the one we wish to discuss here, is philosophical. Types are more than just a technical device; they also capture

meaning. This entire thesis is based on the philosophical position that type expressions retain some meaning even if they don't terminate.

Recursive types have been considered in any number of theories, but the standard theory of recursive types is based on expressions that are *contractive*, which means that they must be values [Pierce, 2002]. For example, $\mu X.\ \mathsf{Int} \to X$ is a recursive or "infinite" arrow-type (it is syntax sugar for $\mathsf{fix}\ \lambda X.\ \mathsf{Int} \to X$). It is meaningful to talk about this expression as a type — it represents the type of a function which can consume an unlimited number of integer arguments. It is also possible to develop complete and decidable algorithms for assigning and comparing recursive types, although such algorithms are not necessarily simple [Colazzo and Ghelli, 1999].

However, the type $\mu X.X$ is not contractive, and it cannot be reduced to a value. What does it mean for a term to have type $\mu X.\ X$? Or, with regards to subtyping, what are the subtypes of $\mu X.\ X$?

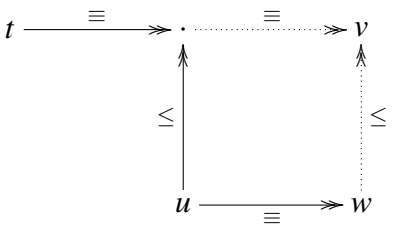
In denotational semantics, a non-terminating expression has the denotation $\bot$, which is the minimal element in a lattice [Tennent, 1976]. Although the lattice model used in denotational semantics is not directly related to subtyping, we might intuitively expect certain similarities. In particular, we might expect every subtype of $\bot$ to be $\bot$ — a non-terminating type should only have non-terminating subtypes.

Our definition of subtyping is syntactic, not denotational. There are a variety of different terms that do not have normal forms, especially if we consider fixpoints, and such terms are not necessarily related syntactically. However, if $\xrightarrow{\equiv}$ and $\xrightarrow{\leq}$ commute, then a similar relationship holds.

Lemma 2.10.1 (Non-terminating supertypes)

Let t be a term which does not reduce to a value: there does not exist a v such that $\Gamma \vdash_A\ t \xrightarrow{\equiv}\!\!\!\!\rightarrow v$. If $\xrightarrow{\equiv}$ and $\xrightarrow{\leq}$ commute, and $\Gamma \vdash_A\ u \leq t$, then u does not reduce to a value either.

Proof: By contradiction. Assume there exists a w such that $u \xrightarrow{\equiv}\!\!\!\!\rightarrow w$. Transitivity elimination gives us $\Gamma \vdash_A u \xrightarrow{\leq}\!\!\!\!\rightarrow \cdot \xleftarrow{\equiv} t$. By commutativity of $\xrightarrow{\equiv}$ and $\xrightarrow{\leq}$, we can derive a v such that $t \xrightarrow{\equiv}\!\!\!\!\rightarrow v$, according to the following diagram. This is a contradiction, so no such w can exist.



□

System $\lambda_{\lhd}$ stands in sharp contrast to other theories of subtyping because soundness does not depend on strong normalization of types. By formulating the subtype relation as an abstract reduction system, we are able to show that type soundness follows from commutativity instead, a property which is similar to confluence. Confluence holds for many systems that are not normalizing, the untyped λ-calculus being a prime example.

Because our notion of soundness is based on commutativity, we conclude that it is possible for a type expression to be meaningful, even if it does not terminate. The practical consequence of this conclusion is the subject of the next chapter, which adds full-blown fixpoints and general recursion over all terms. Of course, the subtyping and well-formedness judgements are obviously undecidable in a theory that admits non-terminating type expressions, but we develop some simple techniques for taming recursion in practice.

2.10.1 Type soundness — an open question

Unfortunately, we cannot conclusively demonstrate that non-terminating type expressions are meaningful, because our proof of type soundness is incomplete. The lack of a complete proof is a major blow to the theory. We can show that System $\lambda_{\lhd}$ will identify type errors at least some of the time, or even most of the time, but we cannot show that it will identify errors all of the time.

The following following term is an example of a type error that will be detected, because there is no derivation of $a \leq \mathsf{Int}$:

$$\lambda f \leq \mathsf{Int} \to \mathsf{Int}.\ \lambda a \leq \mathsf{Top}.\ f(a)$$

Tools that can identify coding errors some of the time do have a place in a programmer's toolkit. The "lint" program, which detects suspicious constructs in C code, is one example of this sort of tool; there are many others. However, the main appeal of

static type checking is that the compiler can identify a certain class of errors all of the time. Moreover, in addition to detecting errors, type information is the basis for many compiler optimizations. Such optimizations are only safe if there is a hard guarantee that type information is accurate. If the type system is unsound, then type information cannot be trusted, and it is much less useful as a result.

Nevertheless, although we do not have a proof that System $\lambda_{\lhd}$ is sound, we do not have a proof that it is unsound either. In the absence of either a proof of soundness, or a counter-example, the soundness of System $\lambda_{\lhd}$ is an open question. We conjecture that it is sound.

The next two chapters introduce the DEEP-- and DEEP calculi, both of which are built on top of System $\lambda_{\lhd}$. Since the metatheory of System $\lambda_{\lhd}$ is already intractable, the meta-theory of DEEP-- and DEEP is even more so. It may seem like an odd decision to build a more complex theory on top of one that is already intractable. However, we did not develop these calculi in the chronological order in which we present them here. We developed the ideas for DEEP first, and only later simplified them to create System $\lambda_{\lhd}$.

Regardless of whether System $\lambda_{\lhd}$ is sound or not, the answer to the question of its soundness has important ramifications. If System $\lambda_{\lhd}$ is sound, then the typing relation is truly unnecessary; subtyping can be used as a substitute for typing. If System $\lambda_{\lhd}$ is unsound, then this fact will likely have ramifications for other type theories that make heavy use of subtyping, as discussed below.

2.10.2 Relevance to other type theories

It is tempting to conclude that the meta-theoretic difficulties which plague System $\lambda_{\lhd}$ stem from the fact that it unifies types and objects. However, that conclusion would be incorrect. The problems found in System $\lambda_{\lhd}$ can be found in any type system which has the following three elements:

1. Type operators with bounded quantification.
2. Higher order subtyping (i.e. point-wise subtyping between operators).
3. A language of types which is not strongly normalizing.

For example, System $F^{\omega}_{\leq}$ is widely used to model inheritance in object-oriented programming languages [Compagnoni and Pierce, 1996]. General-purpose OO languages have fixpoints. Adding fixpoints by themselves to the level of objects in System $F^{\omega}_{\leq}$

does not create any problems, because types are completely separate from objects. However, if one were to add both fixpoints and dependent types, then the existing proof of type safety for System $F^{\omega}_{\leq}$ would break down, because dependent types are not strongly normalizing in the presence of fixpoints.

This is an important result, because a number of researchers are interested in adding dependent types to object-oriented languages in various ways. We predict that future research will either face the same meta-theoretic difficulties that plague System $\lambda_{\triangleleft}$, or be forced to make certain compromises in the type theory. Possible compromises are: (1) placing restrictions on subtyping, or (2) placing restrictions on dependent types.

2.10.3 Relevance to DEEP

In the design of the DEEP calculus, we were unwilling to make such restrictions. As will be shown in chapter 4, higher-order subtyping with bounded quantification provides an elegant model of mixin-based inheritance. A mixin is a function from classes to classes, i.e. a type operator. Higher-order subtyping between mixins provides a clean model of multiple inheritance, a subject that has confounded object-oriented languages for some time.

The second restriction is also unpalatable. As discussed in chapter 1, one of the design goals of the DEEP calculus is to solve the tag-elimination problem. A language with solid support for dependent types makes this problem much easier to solve. As will be discussed in chapter 3, dependent types are also important for modeling first-class modules and virtual classes.

The trouble with dependent types is that recursion at the object level "infects" the language of types. It should be noted that there are some alternative techiques which could potentially address this issue. For example, reduction can be split into object-level and type-level reductions which are analyzed independently [Aspinall and Compagnoni, 1996], or dependencies can be restricted to inductive data types for which termination is ensured [McBride and McKinna, 2004] [Xi, 1998]. We have chosen not to explore such techniques, because even if the language of types could be suitably shielded from recursion at the object level, we would still need to deal with recursion at the type level.

We wish to support recursive type definitions because object-oriented class hierarchies make heavy use of mutually recursive classes. Moreover, as will be explained in chapter 3, DEEP supports *late binding* for types. The easiest way to implement late

binding is to associate fixpoints with modules, and to use a single fixpoint for both object and type definitions within a module. Type-level recursion and object-level recursion cannot be easily separated within this framework.

2.10.4 Summary of main contributions

This chapter introduces "pure subtype systems", a class of type systems which do not distinguish between types and objects, and which are based on subtyping, rather than typing. We believe that this work makes two main contributions to the current state of the art.

First, we have identified a symmetry between typing and subtyping in existing type theories, most notably System $F_{\leq}$ and $F^{\omega}_{\leq}$. System $\lambda_{\lhd}$ demonstrates that it is possible to exploit this symmetry and unify typing and subtyping into a single relation. We have shown that the subtype relation is strong enough to completely subsume typing by embedding System $F_{\leq}$ and System $\lambda*$, the pure type system with $* : *$, into System $\lambda_{\lhd}$.

Second, we have formulated the subtype relation as a reduction system. This is an important step, because there is a rich body of work on reduction systems in the literature, and that work can then be applied directly to the meta-theory for System $\lambda_{\lhd}$. In particular, we show that if the reduction system has a confluence property (commutativity of $\xrightarrow{\leq}$ and $\xrightarrow{\equiv}$), then transitivity can be eliminated, and if transivity can be eliminated, then the type theory is sound.

The main limitation of this work is the fact that our current proof is incomplete. We have shown that subtype reduction commutes locally, but we have been unable to show that it commutes globally. Section 2.9 identifies possible avenues for future research in this area.

Chapter 3

Modules, inheritance, and recursion

Let me introduce you to General Recursion, the general who sends his troops off into battle, without ever caring if they return.

— Connor McBride, AFP Summer School, 2004

3.1 Introduction

The DEEP calculus extends System $\lambda_{\triangleleft}$ with support for modules, inheritance, and recursion. The basic ideas behind DEEP, including the unification of types and objects, the module system, and the emphasis on partial evaluation, were initially developed informally as part of the Ohmu language Hutchins [2003]. These ideas were then formalized into an earlier version of DEEP calculus [Hutchins, 2006], which has been refined into the version presented here.

The name DEEP comes from the fact that the calculus was designed to support *deep mixin composition* [Zenger and Odersky, 2005], a process whereby inheritance on a containing module can be used to recursively specialize nested modules down to any depth. Deep mixin composition is the fundamental technique that allows us to solve the expression problem.

This chapter introduces the DEEP-- calculus, which is a simplified version of DEEP. The syntax of DEEP-- is very similar to DEEP, and the basic concepts are the same, but the type system is significantly simpler. This chapter therefore serves two purposes: it introduces the basic concepts of DEEP within the context of a simpler system, and it describes why the simpler system is adequate for certain kinds of problems, but not others.

Modules present two challenges which motivated the design of DEEP. First, modules contain both types and objects, as members of a single namespace with mutually recursive definitions. In this situation, maintaining a meaningful separation between types and objects is highly non-trivial, even if the underlying type system does have a strong phase distinction [Crary et al., 1999] [Dreyer et al., 2001]. In the presence of dependent types, such separation becomes all but impossible. (See Chapter 5 for more details.)

Second, classes and modules naturally give rise to a very rich notion of subtyping or subclassing. Object-oriented inheritance between classes is the most obvious and well-known example, and a great deal of work has been done on extending the mechanism of inheritance from individual classes to modules and class hierarchies [Ernst, 2001] [Ernst, 2003] [Nystrom et al., 2005] [Nystrom et al., 2006] [Jolly et al., 2005] [Igarashi et al., 2005] [Clarke et al., 2007]. Although subtyping is not as popular in the functional programming community, similar ideas can also be found in functional languages, such as Haskell type classes [Wadler and Blott, 1989], and subtyping for ML modules [Mitchell et al., 1991] [Lillibridge, 1996].

The techniques presented in the last chapter for System $\lambda_{\lhd}$ are directly related to these two challenges. DEEP-- does not distinguish between types and objects, and the type system is based firmly on subtyping rather than typing.

3.1.1 What is a module?

Unlike the word "function", the word "module" has no generally accepted meaning. A number of different module calculi have been proposed in the literature, and they vary greatly in philosophy, complexity, and expressive power. Modules in DEEP-- occupy one point in a fairly large design space, and were developed with the following goals in mind:

Modules may contain both types and objects. As mentioned previously, modules can contain both types and objects as members. This is the primary feature that distinguishes modules from ordinary records, which are found in any number of languages.

Modules are recursive. A module consists of a set of named declarations, which may be mutually recursive. Modules support the definition of both recursive types (e.g. List), and recursive functions or objects (e.g. fold, map etc.). There is no need for separate notions of recursion, such as μ-types or a letrec operator.

Modules are hierarchical. Modules are hierarchically structured; a module may contain nested modules. (The word "nested module" in this document is used in preference to the word "sub-module" to avoid confusion with subtyping.) Hierarchical structure is an important part of scalability; it allows programmers to organize programs of arbitrary size. Large modules are constructed from smaller modules, which are constructed from yet smaller modules, and so on.

Modules are extensible. It is possible for one module to extend another module, using a mechanism that closely resembles OO inheritance. A derived module may override or extend definitions in a base module. Most importantly, overriding is not restricted to methods. DEEP-- allows nested modules and type members to be extended or overridden as well, a process that we call *deep mixin composition*.

Modules are first-class. Modules are ordinary terms in the language, which means that they can be stored in data structures, passed as arguments to functions, and returned as the results of computations. This is a significant philosophical difference from some other module calculi in the literature, which treat the module calculus as a separate linking system that is layered on top of a different core language [Ancona and Zucca, 2002] [Harper and Stone, 2000]. Modules in DEEP-- are an integral part of the core language, and can be used for small structures, such as records, letrecs, and the "objects" of object-oriented programming, as well as large structures.

Modules support separate compilation. Every module has both an interface and an implementation. The implementation is hidden behind the interface, so that changes to the implementation do not affect other modules, or even other definitions within the same module.

Modules are translucent. A *transparent* module is one in which the implementation is completely visible. An *opaque* module is a "black box" in which the implementation is completely hidden. A *translucent* module may expose some parts of the implementation, while hiding other parts [Lillibridge, 1996].

In particular, it is often necessary to export the exact identity of type members for the purpose of static type checking [Stone, 2000] [Leroy, 1994]. The DEEP-- language extends translucency to object members as well, thus providing a model for some common optimizations, such as inlining of final methods. Subtyping is an excellent technique for dealing with translucency, because it provides a smooth spectrum of type information between completely transparent, and completely opaque modules.

3.1.2 Main contribution: extensible modules

The main technical contribution of DEEP--, in comparison to other module systems in the literature, is extensibility. DEEP-- provides a type system that we believe to be statically safe, and which allows modules to be extended, *including nested modules and type members*. Mainstream object-oriented languages and functional languages both lack this capability.

The extension mechanism provided by DEEP-- closely resembles object-oriented inheritance. OO inheritance allows a class to be extended by adding new methods and fields, or by overriding existing methods. Many OO languages, including Java and C++, also support nested or inner classes. Such classes act as both nested modules, and type members of the enclosing class. However, mainstream OO languages, such as Java, only allow overriding of methods; it is not possible to use inheritance to extend or refine a nested class.

The desire to extend mutually recursive nested class declarations has been the basis for a number of experimental languages, and has spawned a great deal of discussion in the literature. The technique is known by various names, including *virtual classes* [Madsen and Møller-Pedersen, 1989], *family polymorphism* [Ernst, 2001], *higher-order hierarchies* [Ernst, 2003], and *feature-oriented programming* [Batory et al., 2003]. In the functional programming community, it has been studied in the form of *mixin-modules* [Duggan and Sourelis, 1996].

The main roadblock to effective implementation of these techniques has been the type system. Extensible recursive modules are difficult to model in existing type theories, and this has prevented widespread use of these techniques in statically typed languages. Chapter 5 compares several alternative solutions in more detail. Suffice it to say that there is a tradeoff between complexity and power; it is hard to create a static type system that is simple, comprehensible, and still strong enough to handle extensible recursive modules.

The DEEP-- and DEEP calculi are one attempt to manage this tradeoff. The distinction between types and objects, and between typing and subtyping, has been jettisoned in the name of simplicity. The immediate cost of this decision, especially in the presence of recursion, is loss of decidability. Our approach in this chapter will be to develop techniques for managing non-termination in practice, rather than attempting to eliminate it altogether.

3.1.3 Outline of chapter

The rest of this chapter is organized as follows:

Section 3.2 provides an overview of how recursion interacts with the type system. Subtyping recursive structures is technically challenging, and there are a number of different approaches and tradeoffs which can be made.

Section 3.3 introduces the surface syntax and operational semantics for the DEEP-- calculus.

Section 3.4 provides examples of how the calculus can be used to solve various typing problems.

Section 3.5 provides a translation from the surface syntax to an underlying model which is used to define the declarative subtype rules.

Section 3.6 discusses the decidability of subtyping, and shows how subtyping is related to partial evaluation.

Section 3.7 discusses some limitations of the system, along with examples of static typing problems that DEEP-- cannot handle.

3.2 Recursion

A recursive structure is one which is self-referential. Such structures are ubiquitous in mainstream programming languages, where they appear as recursive types, recursive functions, and circular or "infinite" data structures. The following code shows three examples in the Haskell programming language:

```
-- recursive type
data List a = Nil
            | Cons a (List a)

-- recursive function
map Nil           = Nil
map f (Cons x xs) = Cons (f x) (map xs)

-- recursive data structure: an ``infinite'' list of 1s
ones = Cons 1 ones
```

Haskell, like every other mainstream programming language, uses general recursion to define structures of the sort shown above. In Haskell, any named definition, including both types and objects, can refer to itself by name within the body of the definition. Because Haskell is a lazy language, its syntax for recursive definitions is particularly uniform.

However, this uniform syntax is something of an illusion, because it masks a number of important decisions that a type theorist must make. There is no one way to implement recursion; instead, there are a variety of different approaches and tradeoffs. This section discusses three questions about recursion that are central to the design of DEEP--:

1. How does recursion interact with dependent types?
2. When are recursive variables bound? Is binding "early", "delayed", or "late"?
3. How is the subtype relation defined over recursive structures?

3.2.1 General recursion and dependent types

General recursion is a double-edged sword. On the one hand, it is simple, convenient, intuitive, flexible, and powerful. On the other hand, general recursion is mathematically problematic. Programs may not terminate, certain forms of program analysis may be undecidable, and it renders the type system inconsistent as a logic [Girard et al., 1989].

A particular problem within type theory is the fact that any attempt to combine general recursion with full dependent types results in a typing judgement that is undecidable. This is one of the main reasons why dependent types are seldom used in mainstream languages.

To see why, assume we have a type Array(T,n), which represents arrays of type T and size n. Is Array(Int, 256) $\equiv$ Array(Int, $4*64$)? The only way to answer this question is to evaluate 4*64. In this case, evaluation terminates because $4*64$ is a constant arithmetic expression. In the general case, however, the type system may be asked to evaluate arbitary object expressions, and such evaluation may not terminate.

The alternative to general recursion is primitive recursion. Primitive recursion does not allow true self-reference, so none of the definitions shown in the Haskell example above would be valid. List and map can be formulated in System F using the well-known Church encodings, an alternative syntax which restricts them to lists of finite size [Pierce, 2002]. Surprisingly enough, circular or infinite data structures like ones can also be encoded in System F using a dual construction based on existential types [Wadler, 1990].

Unlike true recursive types, these System F encodings are restricted; the recursive type variable can only appear in positive positions [Wadler, 1990]. Moreover, all operations on recursive data types must be expressed as folds or unfolds of some kind.

Not only is this notation less convenient for practical programming, but some simple operations like finding the predecessor of a natural number, or adding an element to the head of a stream, are no longer constant time.

Within dependent type theory, *inductive data types* are widely used instead of Church encodings to make primitive recursive types more flexible [Luo, 1994]. However, real-world programs make use of a variety of data structures that cannot be inductively defined; examples include streams, circular lists like ones, doubly-linked lists, and graphs of various kinds. Mainstream programming languages, without exception, are based on true recursive types and general recursion.

With this in mind, type systems which support dependent types must make one of three design decisions:

1. Do not allow general recursion in the language at all [McBride and McKinna, 2004].
2. Allow dependent types and general recursion, but restrict the object expressions that are valid for use in types [Xi, 1998] [Odersky et al., 2003] [Ernst et al., 2006].
3. Give up on decidable type checking [Augustsson, 1998].

Option 1 is the purists choice, but it is of limited practical interest since it cannot be easily applied to mainstream languages. Option 2 would seem to offer the best of both worlds, but the devil is in the details; what restrictions are appropriate, and how should they be enforced?

The DEEP-- language takes option 3, which is the pragmatists choice. It deliberately sacrifices decidability in favor of convenience, simplicity, power, and expressiveness. Instead of preventing non-termination altogether, our goal is to ensure that non-termination is predictable. In other words, there should be straightforward guidelines for writing programs so that type-checking will terminate, and if the compiler does hang, then it should hang for a reason that is (hopefully) obvious to the programmer, and which can be corrected by modifying the program. This issue is discussed in more detail in Section 3.6.

3.2.2 Binding times and fixpoints

Recursive structures are generated mathematically by finding the fixpoint of a *generating function*. For example, the definition of ones can be written in the simply typed

lambda calculus as:

$$\mathsf{ones} = \mathsf{fix}\ \lambda x : \mathsf{List(Int)}.\ \mathsf{Cons}(1, x)$$

Rather than refering to itself by name, the body of ones refers to itself using the intermediate variable x. This intermediate variable will be referred to as a *self-variable* throughout the rest of this document. The fix operator "ties the knot" by binding x according to the following reduction rule:

$$\mathsf{fix}\ \lambda x : T.\ t \longrightarrow [x \mapsto \mathsf{fix}\ \lambda x : T.\ t]t$$

3.2.2.1 Early versus delayed binding

The definition of ones shown above is an example of *early binding*. The self-variable x is bound immediately, at the point at which it is declared. Early binding is well-understood in the theory of recursion, but it has a significant limitation. Because the self-variable is bound immediately, early binding prevents a recursive structure from being modified or extended.

In object-oriented languages, a derived class may override methods in the base class. If method names were bound immediately at the point where the class was declared, then such overriding would not be possible. Consider the following example in Java:

```
class A {
  int foo(int n) { return 1 + bar(n-1); }
  int bar(int n) {
    if (n == 0) return 0; else return foo(n);
  }
};

class B extends A {
  int foo(int n) { return 1 + super.foo(n); }
};
```

In this somewhat contrived example, class A uses two mutually recursive functions to count to n. Class B inherits bar unchanged, but it overrides foo so that it counts to $2 * n$ instead. To make matters more complicated, the new definition of foo not only overrides the old, it calls the old version using the **super** keyword; this is a common pattern in OO languages [Gosling et al., 2005].

In order to capture this process, we must change the point at which the fix operator is applied. Here's the example again, in an untyped λ-calculus with records:

```
Agen = λx. {
  foo = λn. 1 + x.bar(n);
  bar = λn. if (n == 0) then 0 else x.foo(n−1);
}
A = fix Agen;

Bgen = λx. {
  foo = λn. 1 + Agen(x).foo(n);
  bar = Agen(x).bar;
}
B = fix Bgen;
```

This definition separates (Agen), which is the generating function for A, from the recursive structure A. The generating function for B then "inherits" bar by calling Agen, and passing its own self-variable *x* as an argument. It uses the same technique to deal with **super**.

I will refer to this technique as *delayed binding*. The definition of A is exactly the same as it would have been using early binding, but the generating function is now accessible, because the application of the fix operator has been "delayed."

3.2.2.2 Late binding

Delayed binding allows a recursive structure to be extended, but it is somewhat inconvenient. The class A, and the generating function for A, are completely different objects with completely different types. The function Agen is extensible, but not recursive, while A is recursive, but not extensible.

Late binding can be used to create a single object that is both recursive and extensible [Abadi and Cardelli, 1996a]. Whereas early binding uses fix, late binding uses self-application, which we will illustrate by defining an unfold operator. The fix operator computes a recursive structure all at once. In contrast, the unfold operator unrolls a generating function just enough to perform a computation, but never computes the actual fixpoint. For example:

```
unfold(x) = x(x);

A = λx. {
  foo = λn. 1 + unfold(x).bar(n);
  bar = λn. if (n == 0) then 0 else unfold(x).foo(n−1);
}

B = λx. {
  foo = λn. 1 + A(x).foo(n);
  bar = A(x).bar;
```

}

A generating function is unfolded by applying it to itself, as shown in the definition of unfold above. Just as before, a derived class can also inherit from a base class by passing its own self variable as an argument.

The main difference between this example and the previous example is that in the previous version, x represents a record within the body of A and B. A method call thus involves a direct application of the dot-operator, e.g. x.foo(n-1). In this version, x is bound to a generating function instead, and that function must be unfolded to yield a record. Each method call thus involves an unfolding step, e.g. unfold(x).foo(n-1). This technique is refered to as "late binding" because binding the self-variable x is delayed as late as possible — it does not happen until just before a method is actually called. In object-oriented languages, late binding is implemented by passing the value for this as a hidden argument to every method call [Abadi and Cardelli, 1996a] [Gosling et al., 2005].

The DEEP-- calculus uses the unfolding technique shown here, although the syntax of DEEP-- is somewhat different. The main challenge in using this technique is establishing type rules that are safe; the above example is obviously untyped. The type rules for self-application of generating functions are different from the rules for application of ordinary functions.

3.2.3 Subtyping recursive types

Recursive types are very similar to recursive objects in many ways, except that there is a subtype relation defined over types. The exact way in which subtyping is done depends on whether binding is early, delayed, or late. For the purpose of discussion, we will cover all three cases here.

3.2.3.1 μ-types with early binding

Recursive types are often specified in the literature using a μ binder, where $\mu X.\ T$ is essentially syntactic sugar for fix $\lambda X.\ T$. This interpretation of recursive types corresponds to early binding. With early binding, a recursive type such as $\mu X.\ \mathsf{Int} \to X$ is interpreted as a finite representation of an infinite type. In this case it is the type of a function that can consume an unlimited number of integer arguments.

Subtyping between μ-types is similarly interpreted as a comparison between infinite types. Although μ-types are "infinite", they are regular, so it is still possible (in

some systems) to develop a decidable algorithm for comparing them. However, creating an algorithm that is both complete and correct is not necessarily easy, even for first-order type systems [Colazzo and Ghelli, 1999].

Interpreting μ-types as infinite types has a consequence: $\mu X.T \leq \mu X.U$ only if X occurs only in covariant positions within T and U. This restriction rules out a number of seemingly intuitive uses of subtyping, especially in object-oriented programs. For example, consider the following Java-like pseudo-code:

```
class Bird = μX. {
  void mate(X partner) { }
}

class Penguin extends Bird = μX. {
  void mate(X partner) { ... }
}
```

This is an example of the *binary method problem*, which is a classic problem in object-oriented type systems. [Bruce et al., 1995] The intended meaning is to state that every species of bird mates with members of its own species. If b is a bird, we know that its mating partner is some kind of bird, while if b is a penguin, we know that its mating partner is a penguin.

The above code not is correct because the variable X appears in a contravariant position: namely the argument type for mate. This means that Penguin is not a valid subtype of Bird. In this example, the error is particularly hard to spot because the same variable X appears in both places. Intuitively, we might expect X to mean the same thing within the definition of Penguin as it does in Bird, while in reality, the μ binder with early binding does not operate in this way.

3.2.3.2 Match subtyping

Although Penguin is not a subtype of Bird, it has "the same recursive structure", because X appears in the same position in both types [Cook et al., 1990]. This observation was later formalized by Bruce as *match subtyping*. One recursive type *matches* another if it has the same recursive structure, even if it is not a proper subtype [Bruce et al., 1997]. Match subtyping can be implemented in Java 1.5 using *F-bounded polymorphism*, as shown below.

```
class BirdGen<X extends BirdGen<X>> {
  void mate(X partner) { }
}
```

```
class PenguinGen<X extends PenguinGen<X>> extends BirdGen<X> {
  void mate(X partner) { }
}

class Bird    extends BirdGen<Bird>       { }
class Penguin extends PenguinGen<Penguin> { }
```

Rather than using a μ binder with early binding, the Java code above closely resembles delayed binding. BirdGen and PenguinGen are generating functions which are parameterized by X, and Bird and Penguin then take the fixpoint of those functions. F-bounded polymorphism captures the recursive structure of the types by allowing the bounding type of X to mention X itself [Canning et al., 1989].

Informally, we say that PenguinGen "matches" BirdGen, because PenguinGen$<T>$ is a subtype of BirdGen$<T>$ for any T. However, type operators in Java are not first-class, so it is not possible to talk about the formal relationship between PenguinGen and BirdGen by themselves.

In a system with higher-order subtyping, such as System $F^{\omega}_{\leq}$, match subtyping between two recursive types can be seen as ordinary subtyping between the generating functions for those types, an observation which is originally due to Abadi and Cardelli [Abadi and Cardelli, 1996b]. Going back to our λ-calculus pseudo-code, we might write:

```
BirdGen = λX. {
  void mate(X partner) { }
}

PenguinGen = λX. {
  void mate(X partner) { ... }
}

Bird    = fix BirdGen;
Penguin = fix PenguinGen;
```

In this example, PenguinGen is a subtype of BirdGen. A subtype comparison between two functions compares their bodies, while holding the argument X to be a free variable. Unlike the μ binder, the λ binder allows us to conclude that X refers to the same type within the bodies of both PenguinGen and BirdGen.

Once again, however, delayed binding is not as convenient as it could be. A subtype relationship between generating functions does not extend to fixpoints of those functions, so Penguin is not a subtype of Bird.

3.2.3.3 Covariant self-types

Another problem with delayed binding is that if the generating functions are written out as type operators in System $F^{\omega}_{\leq}$, then the self-types will not be covariant. The reader may have noticed that although bounding type of X is shown in the Java version, it is conspicuously missing from the definitions of PenguinGen and BirdGen in the System $F^{\omega}_{\leq}$ version.

It has been omitted because in System $F^{\omega}_{\leq}$, the bounding type of a type operator is either contravariant, or invariant, depending on which version of $F^{\omega}_{\leq}$ is being used. This restriction is not appropriate for match subtyping. When BirdGen is specialized to PenguinGen, the bounding type of X should be specialized accordingly. In other words, the bounding type should be covariant, rather than invariant.

This invariant/covariant conflict arises because λ-abstractions are doing double-duty. A single constructor (λ) is being used with two different destructors: function application, and fix. These two destructors impose different requirements on the sub-type relation.

Function application applies an ordinary function to an external argument. Since the function varies independently of the argument, it is not safe to specialize the input type. The fix operator is different. It operates on a generating function of type $T \rightarrow T$, and essentially plugs the output of that function back in to the input. The semantics of fix are such that it is safe for the input type of a generating function to vary covariantly with the output type.

Fortunately, this issue can be easily resolved by using two different constructors. In DEEP--, a λ-abstraction is an ordinary function, and is eliminated with function application. A μ-abstraction is a generating function for a recursive structure. In the case of delayed binding, μ would eliminated with fix. DEEP-- uses late binding, so μ is eliminated by self-application.

3.3 The DEEP-- calculus

The design of DEEP-- has been heavily influenced by object-oriented languages. However, DEEP-- is a *prototype*-based language, rather than a class-based language. Prototypes were introduced in the Self programming language as an alternative to classes [Ungar and Smith, 1987].

3.3.1 Prototypes

Prototypes in Self are ordinary objects, which exist at run-time and can be used in computations. Rather than inheriting from a superclass, each object in Self has a parent, which is a pointer to another object. When an object recieves a message, it may be able to handle the message itself. However, if the object cannot handle the message, then it *delegates* that message to its parent. Delegation forwards a message up the chain of parents until it finds a method that can handle the message. Every message has a "self" parameter, which is a pointer to the object that originally received the message. Every method can access the original receiver by means of "self" — hence the name of the language.

In practice, delegation is not very different from standard OO inheritance. Programs in Self typically define a set of "prototypes", which are objects that act more or less like classes. Prototypes serve as repositories for behavior that is shared among a number of instances. Each instance delegates to a prototype.

The novelty of the Self approach lies in the fact that delegation is a relationship beween objects, whereas inheritance is a relationship between classes. The Self language demonstrates that the class/instance dichotomy is not strictly necessary. In Self, the relationship between an "instance" and a "class" is exactly the same as the relationship between a "subclass" and a "superclass"; both instantiation and inheritance can be implemented by means of delegation.

Like Self, DEEP-- relies on delegation as the mechanism whereby one module can extend another. Unlike Self, DEEP-- is statically typed. If a delegates to b, then a is a subtype of b. This notion of subtyping between objects is sensible because a can handle any message that b can handle, by definition.

3.3.2 Surface syntax and operational semantics

The surface syntax and operational semantics of DEEP-- are shown in Figure 3.1. This surface syntax is slightly different from the actual formal syntax, which will be introduced later when we define the type system. Using two different syntaxes is somewhat inelegant, but there is a good reason for it. The basic concepts of DEEP-- are easier to explain using the surface syntax, but for technical reasons, the surface syntax is not suitable for the type system. However, we will show that any program written using the surface syntax can be transformed to one using the formal syntax, and vice versa, and that this transformation preserves the operational semantics.

x, y, z	variable	O ::= def | override	modifier
ℓ, l, m	slot name	$\div$ ::= : | =	virtual/final
s, t, u ::=	terms	v, w ::=	values
x	variable	Top	Top-type
Top	Top-type	$\lambda x \leq t.\ u$	function
$\lambda x \leq t.\ u$	function	μx extends t $\{\overline{l}\ \overline{u}\}$	module
μx extends t $\{\overline{d}\}$	module	$: t$ inline$(\overline{s}) = u$	field
$: t$ inline$(\overline{s}) = u$	field	Γ ::=	contexts
$t(u)$	apply	$\emptyset$	empty context
$t@(u).l$	delegate	$\Gamma, x \leq t$	upper bound
$t\$$	extract	$\lhd$ ::=	type relations
d, e, f ::=	declarations	$\leq$	subtype
$O\ l \div t$	labeled term	$\equiv$	equivalence

Notation:

- $\overline{t}$ denotes a sequence of zero or more terms, separated by commas.
- $\overline{d}$ denotes a sequence of zero or more declarations, separated by semicolons.
- dom$(\overline{d})$ denotes the set of labels in the sequence of declarations $\overline{d}$.
- $e \in \overline{d}$ is true if e appears in the sequence $\overline{d}$.

Equality: Terms are considered to be syntactically equal (i.e. α-equivalent) under renaming of bound variables, plus the following rule:

$$\frac{\overline{d} \text{ is a permutation of } \overline{e}}{\mu x \text{ extends } t\ \{\overline{d}\}\ =\ \mu x \text{ extends } t\ \{\overline{e}\}}$$

Evaluation context:

$C ::= [] \mid C(t) \mid t(C) \mid C@(t).l \mid t@(C).l \mid C\$ \mid \lambda x \leq C.\ t \mid \lambda x \leq t.\ C \mid \mu x \text{ extends } C\ \{\overline{d}\}$
$\mid \mu x \text{ extends } t\ \{\overline{d};\ O\ l \div C;\ \overline{e}\} \mid\ : C \text{ inline}(\overline{s}) = u \mid\ : t \text{ inline}(\overline{s}, C, \overline{s'}) = u \mid\ : t \text{ inline}(\overline{s}) = C$

$$\frac{t \longrightarrow t'}{C[t] \longrightarrow C[t']} \qquad \text{(E-CONG)}$$

$$(\lambda x \leq t.\ u)(s) \longrightarrow [x \mapsto s]u \qquad \text{(E-APP)}$$

$$\frac{O\ l \div u \in \overline{d}}{(\mu x \text{ extends } t\ \{\overline{d}\})@(s).l \longrightarrow [x \mapsto s]u} \qquad \text{(E-DLG1)}$$

$$\frac{l \notin \text{dom}(\overline{d})}{(\mu x \text{ extends } t\ \{\overline{d}\})@(s).l \longrightarrow t@(s).l} \qquad \text{(E-DLG2)}$$

$$(: t \text{ inline}(\overline{s}) = u)\$ \quad \longrightarrow \quad u \qquad \text{(E-EXT)}$$

Figure 3.1: The DEEP-- calculus — surface syntax and operational semantics

DEEP-- extends System $\lambda_{\triangleleft}$ with *modules* and *fields*. A module is constructed with the syntax: μx extends t $\{\overline{d}\}$, where $\overline{d}$ is a sequence of declarations, separated by semicolons. Modules are eliminated by *delegation*, using the syntax $t@(u).l$.

A field is constructed with the syntax : t inline$(\overline{s}) = u$. Fields are eliminated by *extraction*, using the syntax $t\$$. Fields and extraction are a technical device which is used by the type system for static typing. They do not have any computational effect.

3.3.3 Modules

A module μx extends t $\{\overline{d}\}$ is composed of three parts:

1. The μx binder declares a self-variable x for the module.
2. The extends t clause declares t to be the parent module.
3. The $\{\overline{d}\}$ syntax defines a record, which consists of a sequence of labeled declarations called *slots*.

A module is a generating function for a recursive structure, as was discussed in the previous section. The self-variable x serves the same role as the **this** keyword in C++ or Java. Slots in a module may be mutually recursive, referring to each other by means of "self". The use of a variable, rather than a hard-coded name like **this**, eliminates ambiguity in the presence of nested modules.

A module will delegate any messages that it does not understand to its parent t. The scope of x includes $\overline{d}$, but not t; it is not possible for a module to inherit from itself, or from any member of itself. A module which does not have a parent may specify Top for t. As a bit of syntactic sugar, we assume that the parent is Top if no parent is specified.

3.3.3.1 Declarations

Declarations have the syntax $O\ l \doteqdot u$, which defines a term u with name l. A declaration can be either a virtual binding $O\ l : u$, or a final binding $O\ l = u$, where l is the name of the slot. Virtual bindings can be overridden in derived modules, while final bindings cannot be.

The modifier O may be either def, which states that the declaration introduces a new slot l, or override, which states that the declaration overrides the definition of l that the module inherits from its parent. If l is a new slot, then l cannot be defined in the parent module, and if l is overriding, then it must be defined in the parent module. As

a bit of syntactic sugar, we assume that the modifier is def if no modifier is specified.

We use an explicit override keyword for two reasons. First, it avoids the "accidental override" problem that sometimes occurs in practical programming. An "accidental override" occurs when a programmer overrides a method without intending to, because some superclass, unbeknownst to the programmer, happens to define one with the same name. Such situations can be avoided by requiring programmers to state their intentions clearly in code. Second, and more importantly, the override keyword introduces a type constraint: an overriding slot must be a subtype of the one defined by the parent. In DEEP--, this constraint could be inferred without requiring a dedicated keyword, but the same is not true of DEEP, as shall be discussed in Chapter 4.

3.3.4 Delegation

The syntax $t@(u).l$ will project the slot l from the module t, by substituting the term u for the self-variable of the module. In the common case, t and u are the same term; the standard OO dot-notation $t.l$ is syntactic sugar for $t@(t).l$. The expression $t.l$ thus projects the slot l from the module t, passing t as the value for "self". This semantics implements late-binding by self-application, as described in the previous section.

Although standard OO dot-notation is adequate in most cases, it is not sufficient to handle inheritance. When a derived module delegates a message to its parent, it must pass its own self-variable as the "self" argument. The full syntax of delegation (i.e. $t@(u).l$) is intended to be used only in this situation.

If a module t does not have a slot named l, then $t@(u).l$ will automatically delegate the message l to the parent of t. Method lookup proceeds up the chain of parents until it finds a matching slot, as is standard practice in OO languages. Slots in derived modules must override slots in their parents which have the same name.

The process of delegation and method lookup is illustrated as follows:

```
A = μx extends Top {
  def a: Int;          // virtual slot
  def b = 2;
  def c = x.a + x.b;
};

B = μx extends A {
  override a = 1;      // override a with final slot
};

B.c ⟶ A@(B).c        // delegate c to parent
    ⟶ B.a + B.b      // substitute B for x
```

```
⟶» 1 + A@(B).b // project a, delegate b to parent
⟶ 1 + 2        // project b
⟶ 3            // add
```

This example defines a module A with three slots: a, b, and c. The value of c depends on a and b via the self-variable x. Module B extends A, and it specializes slot a from Int to 1. Late binding ensures that the change to a is reflected in c; B.c reduces to 3.

3.3.4.1 Delegation and subtyping

The subtype rule for delegation is almost identical to the rule for function application, as shown below:

$$\frac{\Gamma \vdash\ t \leq t', \quad u \equiv u'}{\Gamma \vdash\ t(u) \leq t'(u')} \qquad \frac{\Gamma \vdash\ t \leq t', \quad u \equiv u'}{\Gamma \vdash\ t@(u).l \leq t'@(u').l}$$

This similarity is no accident: the μ binder and the λ binder both define functions (albeit different kinds of functions), and they are both eliminated by variable substitution. The self-argument for delegation is invariant in subtypes, just like the argument for ordinary function application. Invariance on arguments is required because a variable may occur in an invariant position within the body of function, or within the body of a slot.

In the case of OO dot notation, the above rule can be simplified to the following:

$$\frac{\Gamma \vdash\ t \equiv u}{\Gamma \vdash\ t.l \leq u.l}$$

This rule is somewhat more restrictive than what one might intuitively expect. Indeed, it is more restrictive than some other proposed type systems for virtual classes [Clarke et al., 2007]. Intuitively, one might expect that if $t \leq u$, then $t.l \leq u.l$, but it is not hard to construct a counter-example that would break this assumption:

```
A = μx {
  T: Top;
  FunType: x.T → x.T;
};

B = μx extends A {
  override T: Int;
};
```

The above code is legal in DEEP--. Module A has two type members: T and FunType. A.FunType yields Top → Top, while B.FunType yields Int → Int. Although B is a subtype of A, B.FunType is not a subtype of A.FunType, because the definition of FunType uses the variable x in an invariant position: the argument type of a function.

3.3.4.2 Well-formedness of delegation

Let M be a module, where $M = \mu x$ extends t $\{\overline{d}\}$. Modules are eliminated by self-application, so the type bound for x is M. The self-variable x can thus be bound to M itself, or to any subtype of M. The well-formedness rule for delegation reflects this requirement:

$$\frac{\Gamma \vdash u \leq_{\text{wf}} t \quad l \in \text{dom}(\overline{d}) \qquad \Gamma \vdash t \leq_{\text{wf}} \mu x \text{ extends } s\ \{\overline{d}\}}{\Gamma \vdash t@(u).l \text{ wf}}$$

The above rule states that $t@(u).l$ is well-formed only if u is a subtype of t. This restriction allows the common case of self-application, where $u = t$, and it allows derived modules to delegate to base modules. The rule also checks that the slot name l is defined in some supertype of t.

3.3.5 Declarations and Fields

Fields are a technical device which is used to provide precise control over inheritance. They are called "fields" because they serve a similar purpose to fields in OO languages such as Java. The purpose of fields can be illustrated in the following example. (Note that for simplicity, we have omitted the inline clause; this is syntactic sugar for the empty clause: inline().)

```
A = μx {                B = μx extends A {          C = μx extends B {
  a: Top;                 override a: Int;            override a = 1;
  b: Int;                 override b = 2;
  c: (:Int = 3);          override c: (:Int = 4);     override c: (:Int = 5);
};                      };                          };
```

This example defines a module A that contains three slots, a, b, and c. Although all three slots contain integers, they are constrained in three different ways. The constraints on each slot determine how they are allowed to vary in derived modules, such as B and C.

Virtual bindings: The declaration a: Top is a *virtual binding*. A derived module can specialize a virtual binding to any subtype. In the sequence above, a is specialized from Top to Int to 1, while b is specialized from Int to 2.

Final bindings: The declaration b = 2 is a *final binding*. A derived module is not allowed to specialize a final binding. Once b has been set to 2 in B, there is no way for C to further refine it. Final bindings for type members are an important part of static type checking. Final bindings for methods and object members are used for common compiler optimizations, most notably inlining.

Fields: The declaration c: (: Int = 3) defines c to be a *field*. A field has two parts: an interface (Int), and an implementation (3). The implementation is constrained to be a subtype of the interface. Unlike ordinary declarations, fields allow overriding: a derived module may override the implementation of a field with any other definition, so long as the interface remains the same.

Overriding is different from specialization. In the example above, the value of c is changed from 3 to 4 to 5. This would not have been possible with virtual or final bindings, because the number 4 is not a subtype of 3. However, (: Int = 4) is a subtype of (: Int = 3). Subtyping between fields only compares their ranges; it does not compare their implementations, so a derived module may override the implementation with something else. The subtyping and equivalence rules for fields are:

$$\frac{\Gamma \vdash\ t \equiv t'}{\Gamma \vdash\ (:t\ \mathsf{inline}(\overline{s}) = u) \leq (:t'\ \mathsf{inline}(\overline{s'}) = u')}$$

$$\frac{\Gamma \vdash\ t \equiv t', \quad u \equiv u', \quad \overline{s} \equiv \overline{s'}}{\Gamma \vdash\ (:t\ \mathsf{inline}(\overline{s}) = u) \equiv (:t'\ \mathsf{inline}(\overline{s'}) = u')}$$

As is standard practice in OO languages, the range of a field is invariant in subtypes. Having an invariant range is not important in DEEP--, but it is important in DEEP; see Section 4.4.1.

3.3.5.1 Fields are first-class

Unlike declarations, fields are first-class terms. Whereas a declaration $O\ l \doteqdot t$ can only appear as a slot within a module, a field can appear anywhere within an expression. The example code in this thesis will commonly put fields within the bodies of functions, e.g.

```
foo: λn ≤ Int. : Int = n + 1;
```

In this particular case, the field could have been shifted to be outside the function:

```
foo: (: Int → Int = λn ≤ Int. n + 1);
```

However, fields are used for more than just overriding, and not all fields can be shifted in this way. As will be discussed in Section 3.6, fields are also used to hide type information. A field $(: t = u)$ hides the implementation u behind the bounding type t. Information hiding is controlled by the inline clause, which must be within the scope of function arguments like n in order to be effective. Section 3.6 is devoted to inlining and partial evaluation; these issues can be safely ignored for the time being.

Although treating fields as first-class terms is an unusual design decision, it simplifies the calculus in certain ways.

3.3.5.2 Extraction

Because fields are ordinary terms, they must be paired with an eliminator. If t is a field, then the expression $t\$$ will extract the implementation of t:

$$(: t \ \mathsf{inline}(\overline{s}) = u)\$ \quad \longrightarrow \quad u$$

Extraction is invariant. If t and u are fields, then $t \leq u$ does *not* imply that $t\$ \leq u\$$. If one field is a subtype of another, then their interfaces are related, but their implementations may not be. The only valid subtype rule for extraction is simple equivalence. (This is another way in which fields differ from declarations.)

$$\frac{\Gamma \vdash\ t \equiv u}{\Gamma \vdash\ t\$ \equiv u\$}$$

Fields and extraction have no computational effect. A field places a type constraint on a term, and extraction eliminates that constraint to yield the original term. Fields and extraction are used by the type system to track types with greater precision at compile time, but they can be safely erased from compiled code.

3.3.5.3 Abstract fields

An abstract field is a field without an implementation, and is written using the syntax $(: t = \_)$. Abstract fields are analagous to abstract methods in Java. The implemetion will be supplied by some derived module which overrides the field.

In the core calculus, abstract fields are syntax sugar for $(: t = t)$. Because DEEP-- uses subtyping rather than typing, and $t \leq t$ by definition, it is safe (although not necessarily meaningful) to use the interface of the field as a placeholder for the missing

- The standard OO dot-notation $t.l$ is sugar for $t@(t).l$.
- The expression $\mathsf{super}_x.l$ is sugar for the term $t@(x).l$ if it appears within the context: μx extends t $\{...\mathsf{super}_x.l...\}$.
- A record without a parent, $\mu x\{\overline{d}\}$, is sugar for one which extends Top, μx extends Top $\{\overline{d}\}$.
- If a declaration has no modifier, e.g. $l = t$, then the modifier is assumed to be def, e.g. def $l = t$.
- A field without an inline clause, $: t = u$, is syntax sugar for one with an empty clause, $: t$ inline$() = u$.
- A field without an implementation $: t = \_$, is syntax sugar for one where the implementation is the same as the range, $: t = t$.
- $t \to u$ is syntax sugar for $\lambda x \leq t.\ u$, where x does not occur free in u.
- A virtual function: $l(\overline{x} \leq \overline{t}) : u$ is sugar for $l : \lambda x_1 \leq t_1.\ ...\lambda x_n \leq t_n.\ u$
- A final function: $l(\overline{x} \leq \overline{t}) = u$ is sugar for $l = \lambda x_1 \leq t_1.\ ...\lambda x_n \leq t_n.\ u$
- Function applications with multiple arguments, $f(a_1,...a_n)$ are curried $f(a_1)...(a_n)$.
- A virtual field: $l : t = u$ is syntax sugar for $l : (: t = u)$.
- A final field: final $l : t = u$ is syntax sugar for $l = (: t = u)$.
- A virtual method: $l(\overline{x} \leq \overline{t}) : u = s$ is sugar for $l : \lambda x_1 \leq t_1.\ ...\lambda x_n \leq t_n.\ : u = s$.
- A final method: final $l(\overline{x} \leq \overline{t}) : u = s$ is sugar for $l = \lambda x_1 \leq t_1.\ ...\lambda x_n \leq t_n.\ : u = s$.
- The extraction operator \$ may be omitted when its intended placement is obvious.

Figure 3.2: The DEEP-- calculus — syntactic sugar

implementation. Unlike Java, DEEP-- does not check to make sure that all abstract fields have been implemented properly in derived classes; it has no way of distinguishing between abstract and concrete classes. This is a serious limitation of DEEP--, and is discussed 3.7.

3.4 Examples

This section will illustrate the mechanics of the DEEP-- calculus, as well as its expressive power, through a series of examples that cover both simple and complex typing is-

Predefined data types:

- Bool is the supertype of true and false.
- Int is the supertype of whole numbers, e.g. 1, 5, and -23.
- Float is the supertype of rational numbers, e.g. 1.235 and -4.71.
- String is the supertype of character strings, e.g. "hello".
- List(T) is the supertype of lists which contain elements of type T. A list is either Nil, or Cons(h,t). As in Haskell, list literals can be written using square brackets, e.g. [1,2,3] is syntax sugar for Cons(1, Cons(2, Cons(3, Nil))).

Predefined operations:

- Standard arithmetic and comparison operators, e.g. +,-,∗, &&, ||, ==, !=, <, etc.
- if then else expressions.
- Simple pattern matching over lists: case lst of { Nil −> ...; Cons(x,xs) −> ...;}
- List comprehensions are defined as syntax sugar for map and filter on lists. The syntax and semantics of comprehensions are the same as in Haskell [Hudak et al., 1999], e.g. if lst is a subtype of List(T), then [$t \mid x <-$ lst] is syntax sugar for map($\lambda x \leq$ T. t)(lst).

Example encoding of lists and pattern matching:

```
List(T ≤ Top) =
  λR ≤ Top. λnil_case ≤ R. λcons_case ≤ T → List(T) → R. R;

Nil(T ≤ Top) =
  λR ≤ Top. λnil_case ≤ R. λcons_case ≤ T → List(T) → R. nil_case;

Cons(T ≤ Top, head ≤ T, tail ≤ List(T)) =
  λR ≤ Top. λnil_case ≤ R. λcons_case ≤ T → List(T) → R.
     cons_case(head, tail);
```

If lst $\leq$ List(T), $t \leq$ R, and $u \leq$ R, then:

case lst of { Nil −> t; Cons(x,xs) −> u; } is syntax sugar for:
lst(R, t, λx ≤ T. λxs ≤ List(T). u)

Figure 3.3: The DEEP-- calculus — predefined types and objects

---

sues. For the sake of clarity, Figure 3.2 shows some syntactic sugar that makes DEEP-- programs more readable, and brings the calculus almost to the level of a simple OO language.

In the interest of readability, the extraction operator ($) will be omitted from most

of the example code in this thesis. Although it is necessary as a technical device, it tends to clutter the code. The intended placement of $ is usually obvious. If foo is a method, declared as: foo(n ≤ Int): Int = n+1, then any application foo(a) should be written as foo(a)$.

Furthermore, we assume that the calculus has been extended with basic data types and expressions, as shown in Figure 3.3. These basic data types and operations can be easily encoded in DEEP-- using standard techniques; there is no need to use modules or classes. Figure 3.3 shows an example of how lists with simple pattern matching can be encoded.

By a slight abuse of notation, we present example code as a set of top-level definitions rather than a single term; the precise method of handling such definitions is discussed in Section 3.5.

3.4.0.4 Example of syntactic sugar

Using syntax sugar, the recursive foo/bar example from the previous section looks like the following in DEEP--.

```
A = μx {
  foo(n ≤ Int): Int = 1 + x.bar(n−1);
  bar(n ≤ Int): Int = if (n == 0) then 0 else x.foo(n);
};

B = μx extends A {
  override foo(n ≤ Int): Int = 1 + super_x.foo(n);
};
```

The sugaring here is superficial, and eliminates some of the more arcane aspects of dealing with a core calculus. For comparison, the full, non-sugared syntax is:

```
A = μx extends Top {
  def foo: λn ≤ Int. : Int inline() = 1 + x@(x).bar(n−1)$;
  def bar: λn ≤ Int. : Int inline() =
    if (n == 0) then 0 else x@(x).foo(n)$;
};

B = μx extends A {
  override foo: λn ≤ Int. : Int inline() = 1 + A@(x).foo(n)$;
};
```

The unsugared version makes the following changes. The OO dot-notation x.bar has been replaced with x@(x).bar, and super_x.bar becomes A@(x).bar. OO-style method declarations are explicit λ-abstractions, with the def modifier inserted as necessary.

Empty clauses, such as the parent of A and the empty inline() declarations are no longer omitted. Finally, the code is now decorated with explicit extractions $.

3.4.0.5 Types versus objects yet again

Although the DEEP-- calculus does not distinguish between types and objects in theory, in practice DEEP-- programs will naturally tend to use some terms as types, and some terms as objects. We adopt the following convention to clarify how terms are being used.

Types, classes, and modules will be written using upper-case identifiers, such as A and B in the previous example. Objects and methods will be written using lower case identifiers, such as n, foo, and bar.

In addition, methods and objects will usually be declared using fields, while types and nested modules will be declared with virtual or final bindings. This difference reflects the fact that inheritance is the most useful way to extend a class or module, while overriding is the most useful way to change the implementation of a method. However, this is not a hard and fast rule – there are times when it is useful to override a type member, or specialize an object member.

3.4.1 Classes, instances, and prototypes

Figure 3.4 is our first example. It shows how to define a simple class, how to inherit from a class, and how to instantiate a class. The code defines a class for 2D points, which has a simple method named radius. Point3 extends that class to 3D points by adding a new variable z and overriding the radius method. The function makePoint3 will construct 3D points.

This example also illustrates the prototype model used by DEEP--. The Point3 "class" is not really a class in the traditional sense, it is a prototype, which means that it is not given any special treatment. Both Point3 and p111 are ordinary objects. Point3.radius will return 0.0, while p111.radius will return sqrt(3).

Moreover, inheritance and instantiation are the same operation. The makePoint3 constructor creates an instance of Point3 by using the exact same extends clause that Point3 uses to inherit from Point2. Consequently, the subtype relationships are:

$$\mathsf{p111} \leq \mathsf{Point3} \leq \mathsf{Point2}$$

```
Point2 = μthis {
  x: Float = 0.0;
  y: Float = 0.0;
  radius: Float = sqrt(this.x*this.x + this.y*this.y);
};

Point3 = μthis extends Point2 {
  z: Float = 0.0;
  override radius: Float = sqrt(this.x*this.x + this.y*this.y +
                                this.z*this.z);
};

makePoint3(ix: Float, iy: Float, iz: Float) =
  μthis extends Point3 {
    override x: Float = ix;
    override y: Float = iy;
    override z: Float = iz;
  };

p111 = makePoint3(1.0, 1.0, 1.0);
```

Figure 3.4: Example of simple classes, inheritance, and instantiation

3.4.2 Virtual types

In addition to overriding methods, a derived module may specialize type members. This capability is known in the literature as *virtual types* [Igarashi and Pierce, 1999]. Since DEEP-- does not properly distinguish between types and objects, the word "type member" really means "a member which is used as a type," but the basic principle remains the same.

One of the classic examples of virtual types is the "cow example" [Torgersen, 1998], which is shown in Figure 3.5. This example involves two class hierarchies – one of food, and the other of animals. The task is to make sure that each type of animal is only allowed to eat its own particular type of food.

The definition of feed1 should generate a type error at compile time. It is not possible to feed any kind of food to any kind of animal, because this would allow feeding meat to cows. The DEEP-- type system will correctly reject feed1. The method a.eat requires an argument of type a.FoodType; the term a is substituted for the self-variable *x*.

The exact value of a.FoodType is not statically known because it depends on a, so

```
Food    = μx { ... };
Grass   = μx extends Food { ... };
alfalfa = μx extends Grass { ... };
hay     = μx extends Grass { ... };

Animal = μx {
  FoodType:      Food;
  favoriteFood: x.FoodType;
  eat(f ≤ x.FoodType): Animal = ...;
};

Cow = μx extends Animal {
  override FoodType     = Grass;
  override favoriteFood = alfalfa;
  override eat(f ≤ x.FoodType): Animal = ...;
};

feed1(a ≤ Animal, f ≤ Food): Animal =
  a.eat(f);                 // wrong!

feed2(a ≤ Animal): Animal =
  a.eat(a.favoriteFood);    // ok

feed3(a ≤ Cow): Animal =
  a.eat(hay);               // ok
```

Figure 3.5: Example of virtual types

---

a.FoodType is treated much like a polymorphic type variable. The only way to obtain a value of type a.FoodType is by calling a.favoriteFood. The definition of feed2 is thus correct.

This example also demonstrates why final bindings are important. The definition of Cow uses a final binding for FoodType. If a is a cow, then a.FoodType is statically known, even if the value of a is not. FoodType = Grass for all cows, which means that it is safe to feed hay to a cow. This use of final bindings for types is also an important part of other module systems, where it appears as *manifest types*, and *translucent sums* [Leroy, 1994] [Lillibridge, 1996].

An alternative way to solve the cow problem in Java 1.5 would be to make Animal a generic class, which is parameterized by the type of food it eats, e.g.

```
Animal<FoodType extends Food> { ... }
```

and define Cow so that it extends Animal<Grass>. A solution using generics has the advantage that it does not require dependent types. It has the disadvantage that every type member must be a formal parameter, which could potentially result in a large number of parameters for complex class hierarchies.

3.4.3 Virtual classes and family polymorphism

The next example shows how DEEP-- can be used to extend nested classes or modules, even in cases where the classes are mutually recursive. This example is not so easy to do with generics.

Mutually recursive classes are used extensively in OO class hierarchies, but it is hard to extend such classes using standard OO inheritance. Any attempt to extend the classes individually will break the type relationships between them, thus destroying static type safety, and requiring run-time type casts. The following example is due to Ernst, who coined the term *family polymorphism* to describe situations of this sort. [Ernst, 2001]

A graph consists of a collection of nodes, which are connected by edges. More specific kinds of graph may annotate nodes and edges with various information. A colored graph, for example, associates a color with each node. A graph coloring algorithm should assign colors to nodes in such a way that no node is connected to another node with the same color.

Before presenting the solution in DEEP--, it is helpful to see why a naïve solution in Java, using a flat class hierarchy, is not statically safe. Figure 3.6 shows some code in Java. The problem with this solution is that it is possible to mix colored nodes and non-colored nodes in the same graph. Because a colored graph may contain non-colored nodes, the valid method must insert a run-time cast from Node to ColoredNode.

Although it is possible to solve this problem in Java 1.5 using parameterized classes, the solution involves adding f-bounded node and edge parameters to all five classes. All five classes must parameterized, because all five classes refer to Node and Edge in some way. Not only is such a solution needlessly convoluted, but it simply does not scale. Large class frameworks routinely contain hundreds or even thousands of classes. Although only a minority of such classes might realistically be candidates for future extension, this is still a heavy burden.

A solution in DEEP-- which uses *virtual classes* is shown in Figure 3.7. Node and Edge are encapsulated as nested classes within Graph. When ColoredGraph inherits from

```
class Node {
  List<Edge> edges;
}

class Edge {
  Node from;
  Node to;
}

class Graph {
  List<Node> nodes;
}

class ColorNode extends Node {
  Color color;
}

class ColoredGraph extends Graph {
  boolean valid() {
    boolean result = true;
    for (Node n : nodes) for (Edge e : n.edges) result = result &&
      ((ColorNode) n).color != ((ColorNode) e.to).color;
    return result;
  }
}
```

Figure 3.6: Example of how family polymorphism fails in Java

Graph, it refines the Node class. The new definition of Node inherits from super_x.Node — this is the key mechanism that allows inheritance to be extended to large-scale class hierarchies.

Graph and ColoredGraph define two different type families, which are not interchangeable. References to Node and Edge are written as x.Node or x.Edge; they are parameterized by the self-variable x. This means that Graph.Edge contains a list of Graph.Node, because x is bound to Graph, while ColoredGraph.Edge contains list of ColoredGraph.Node. Since Graph.Node and ColoredGraph.Node are different types, there is no need for any downcasts in the definition of valid.

Refining a nested class is more delicate than specializing a type member, because Node and Edge both refer to x — the self-variable of the enclosing module. A nested module thus has a lexical context that must be handled in an appropriate way. It is important that Node inherits from super_x.Node, which is syntax sugar for Graph@(x).Node,

```
Graph = μx {
  Node: μy {
    edges: List(x.Edge) = _;
  };

  Edge: μx {
    from: x.Node = _;
    to:   x.Node = _;
  };

  nodes: List(x.Node) = _;
};

ColoredGraph = μx extends Graph {
  override Node: μy extends super_x.Node {
    color: Color = _;
  }

  valid: Bool = and([ n.color != e.to.color |
                      n <- x.nodes, e <- n.edges]);
};
```

Figure 3.7: Example of family polymorphism in DEEP--

---

rather than from Graph.Node, which is syntax sugar for Graph@(Graph).Node. Delegation adjusts the self-variable appropriately so that the lexical context is preserved. The type system will reject any program which does not follow this pattern of extension as being ill-formed.

Note that ColoredGraph.Node is *not* a subtype of Graph.Node; if it were, then the type system would not be safe, as explained earlier in section 3.3.4.1. ColoredGraph.Node is a subtype of Graph@(ColoredGraph).Node, not Graph@(Graph).Node. Delegation allows the Node class to be reused in a new type family; it does not establish a subtype relationship with the Node class in the original type family.

3.4.4 Nominal typing with dependent path types

One of the differences between object-oriented type systems and functional type systems is that the former are usually *nominal*, while the latter are usually *structural*. In a nominal system, types have names, and two types are equivalent if and only if they have the same name. In a structural system, types are represented by complex structures (e.g. record, function, or tuple types) rather than by name, and two types are

equivalent if they have the same structure.

DEEP-- blends these two techniques. On the surface, it appears to be a structural system. Types are represented by arbitrary terms, and the subtype rules perform a structural comparison between terms. In practice however, typing often has a more nominal flavor.

A type such as x.Node is a *dependent path type*. [Odersky et al., 2003] [Ernst et al., 2006] A path is a sequence of projections $x.l_1.l_2...l_n$, so-called because it resembles the paths used to access files in a file system. It is dependent because it depends on the object x.

Because x is a variable, and the Node slot uses a virtual binding, it is not possible to reduce x.Node to a module definition. When x.Node is used as a type, it thus behaves very much like a nominal type — all type comparisons are done based on the variable name x, and the slot name Node.

3.4.5 Type classes

Our final examples demonstrate that DEEP-- can be used to represent modules in functional languages as well as classes in OO languages. In particular, we show how Haskell type classes can be encoded within DEEP--. This example is conceptually much simpler than the previous two. One of the advantages of type classes is that methods are defined separately from types, so there is no need for some of the typing heroics that characterize object-oriented systems.

Figure 3.8 shows how type classes can be encoded in DEEP--, based on the standard dictionary-passing mechanism [Wadler and Blott, 1989]. The function max accepts a type T, a dictionary of comparison operators on T, and then two objects of type T as arguments. It then uses the dictionary to compare the two objects. Note that in the core calculus, both the type parameter and the dictionary for that parameter must be passed as explicit arguments; a real implementation would hopefully infer these arguments from the context in some way.

There is one notable flaw with this encoding. Although Ord(T) is a subtype of Eq(T), instOrdInt is not a subtype of instEqInt. We would expect instOrdInt to inherit the implementation of == from instEqInt, but it doesn't; it explicitly copies the implementation. This flaw is due to the fact that DEEP-- does not support multiple inheritance. We will revisit this example in the next chapter, and show how a proper version can be implemented in DEEP; see Section 4.8.1.

```
Eq(T ≤ Top) = μx {
  ==(a ≤ T, b ≤ T): Bool = _;
  !=(a ≤ T, b ≤ T): Bool = !x.==(a,b);
};

Ord(T ≤ Top) = μx extends Eq(T) {
  < (a ≤ T, b ≤ T): Bool = _;
  > (a ≤ T, b ≤ T): Bool = _;
  <=(a ≤ T, b ≤ T): Bool = x.==(a,b) || x.<(a,b);
  >=(a ≤ T, b ≤ T): Bool = x.==(a,b) || x.>(a,b);
};

instEqInt = μx extends Eq(Int) {
  override ==(a ≤ Int, b ≤ Int): Bool = __eqInt(a,b);
};

instOrdInt = μx extends Ord(Int) {
  override ==(a ≤ Int, b ≤ Int): Bool = __eqInt(a,b);
  override < (a ≤ Int, b ≤ Int): Bool = __ltInt(a,b);
  override > (a ≤ Int, b ≤ Int): Bool = __gtInt(a,b);
};

max(T ≤ Top, ord ≤ Ord(T), a ≤ T, b ≤ T): Bool =
  if (ord.>(a,b)) then a else b;

myMaxValue = max(Int, instOrdInt, 3, 4);
```

Figure 3.8: Example of type classes in DEEP--

In a sense, this is a trivial example. The "magic" of type classes is not about passing dictionaries explicitly. The magic (and potential complexity) comes from passing dictionaries implicitly. Haskell will automatically infer which dictionaries are required, and locate the appropriate ones to pass in each case [Peyton Jones et al., 1997] [Lewis et al., 2000].

On the other hand, complex inference is never part of a core calculus, and this example demonstrates two properties that Haskell lacks. First, type classes and their instances are first-class modules; their representation is not hidden behind the machinery of the type system. Second, Ord(T) is a subtype of Eq(T), whereas Haskell uses coercion rather than subtyping.

DEEP-- follows the same philosophy as the "Modular Type Classes" proposed by Dreyer, Harper, and Chakravarty [Dreyer et al., 2007]. They argue that it is better to build an expressive module system first, and then add automatic inference rules for the

```
MapKey(K ≤ Top) = μx {
  Dict(T ≤ Top): Top;  // abstract type constructor

  empty (T ≤ Top): x.Dict(T) = _;
  insert(T ≤ Top, dict ≤ x.Dict(T), key ≤ K, val ≤ T): x.Dict(T) = _;
  lookup(T ≤ Top, dict ≤ x.Dict(T), key ≤ K): T = _;
};

instMapKeyInt = μx extends MapKey(Int) {
  override Dict  (T ≤ Top)           = RadixInt.Dictionary(T);
  override empty (T ≤ Top): x.Dict(T) = RadixInt.makeEmptyDict(T);
  override insert(T ≤ Top, dict ≤ x.Dict(T), key ≤ K, val ≤ T)
    : x.Dict(T) = RadixInt.insertElem(T, dict, key, val);
  override lookup(T ≤ Top, dict ≤ x.Dict(T), key ≤ K): T =
    RadixInt.lookupElem(T, dict, key);
};

insertLookup(K ≤ Top, T ≤ Top, mapk ≤ MapKey(K),
             key ≤ K, val ≤ T): T =
  mapk.lookup(T, mapk.insert(T, mapk.empty(T), key, val), key);
```

Figure 3.9: Example of associated types in DEEP--

common cases, rather than to focus first on inference, at the expense of the module system.

3.4.6 Associated types

Virtual types have been proposed for Haskell, where they are called *associated types* [Chakravarty et al., 2005]. Haskell 98, much like mainstream OO languages, only allows a type class to contain methods. Associated types extend type classes so that they can contain type declarations as well.

The example used by Chakravarty et al is that of a dictionary, which is a mapping from keys to values. An efficient dictionary should use a different data structure depending on the type of key used to index the dictionary. A dictionary indexed by integers may use a different data structure than one indexed by strings. The purpose of associated types is to automatically determine, given a particular key type, what the associated dictionary type for that key should be. Note that while the implementation of a dictionary will differ depending on the key type, the implementation should be fully polymorphic with respect to the element type.

Figure 3.9 shows this example in DEEP--. The data type used to build dictionaries for a key is stored as a member of the type class for that key. In the case of integers, the above code uses radix (i.e. Patricia) trees. A dictionary of strings might choose to use binary search trees instead.

The virtual type operator Dict(T) is completely hidden from clients of the module. The insertLookup function creates a new empty dictionary, inserts a value, and then looks up that value, without ever refering to the dictionary type.

3.5 Formal system

The previous sections introduced the surface syntax for DEEP--, and gave several examples which illustrate how the language can be used to solve various problems. This section introduces the formal syntax for DEEP--, which differs slightly from the surface syntax, and presents the formal subtype and well-formedness rules of the system.

3.5.1 Equational reasoning for recursive structures

The surface syntax is unsuitable for formal proofs because of a well-known problem with recursive structures. Certain forms of reduction on recursive structures are not necessarily confluent, leading to a failure of equational reasoning. [Ariola and Klop, 1997] [Ariola and Blom, 1997] The following example, originally due to Ariola and Klop, illustrates the problem. [Ariola and Klop, 1997]

```
letrec a = b + 1; b = a + 1; in a
```

Intuitively, this letrec could be reduced to two different terms, by substituting for either a or b, respectively:

```
letrec a = a + 1 + 1; b = a + 1;      in a
letrec a = b + 1;     b = b + 1 + 1; in a
```

These two terms are syntactically different. The first term refers only to a, while the second refers only to b. Moreover, no matter how many reductions are performed, the first will always have an even number of additions in a, while the second will always have an odd number. They therefore have no common reduct, which means that the reductions are not confluent.

Note that the above problem does not arise in the untyped λ-calculus, even though it is possible to encode fixpoints, because the exact reductions shown above cannot occur.

A fixpoint-based semantics effectively substitutes for all variables simultaneously. The operational semantics of DEEP-- remain confluent for a similar reason.

However, this issue does affect the type system for DEEP--. The type system must be able to compare two modules to determine if they are equivalent, or if one is a subtype of the other. Algorithmic subtyping, presented for System $\lambda_{\lhd}$ in the last chapter, defines subtyping as a reduction system. If this technique were extended to modules, then the relevant subtype reductions would resemble those shown above.

There are a few ways of circumventing this problem which can be found in the literature. Confluence can be restored by establishing an order on the variables in the letrec, so that substitution in mutually recursive definitions is not allowed [Ariola and Klop, 1997] [Wells and Vestergaard, 2000]. Alternately, Ariola and Blom show that the above reductions satisfy an approximate notion of confluence; the two cases are congruent because they have the same infinite expansion [Ariola and Blom, 1997]. Other approaches show that the two cases are observationally equivalent, or that they satisfy *meaning preservation* [Wells et al., 2003].

However, none of these mechanisms seem to be appropriate for DEEP--. An ordering on slot names is hard to define; the situation in DEEP-- is complex because of delegation, and because the self-variable of a module often occurs bare. A type theory based on infinite expansions or observational equivalence would be extremely difficult to work with.

The formal syntax of DEEP-- sidesteps this problem entirely, by using nominal subtyping rather than structural subtyping for modules.

3.5.2 Nominal subtyping

The formal syntax for DEEP-- is shown in Figure 3.10. The mechanism used to handle modules in DEEP-- is very similar to the one used to handle classes in Featherweight Java [Igarashi et al., 1999].

Modules are declared as top-level definitions in a global module table. A program is written as $\mathsf{let}\ \overline{G} = \overline{M}\ \mathsf{in}\ u$, where $\overline{G} = \overline{M}$ is a mapping from module names G_i to module definitions M_i. To lighten the notation, we assume that the table is fixed and cannot be extended. The notation $GT(G)$ refers to the definition named G in the global table.

The module syntax $\mu x\ \mathsf{extends}\ t\ \{\overline{d}\ \}$ is no longer a term; it can only occur as a top-level definition. Within a term, modules are represented with the syntax $G(\overline{t})$.

x, y, z	variables	Program ::=	program
ℓ, l, m	slot names	let $\overline{G} = \overline{M}$ in t	letrec
G, H	global variables		
		M, N ::=	module definition
s, t, u ::=	terms	μx extends t $\{\overline{d}\}$	module
x	variable	$\lambda x \leq t.\ M$	parameterized
Top	Top-type		
$\lambda x \leq t.\ u$	function	d, e, f ::=	declarations
$G(\overline{t})$	module	$O\ l \div t$	labeled term
$: t$ inline$(\overline{s}) = u$	field		
$t(u)$	apply	O ::= def $\mid$ override	modifier
$t@(u).l$	delegate	$\div$ ::= : $\mid$ $=$	virtual/final
$t\$$	extract		
		Γ ::=	contexts
v, w ::=	values	$\emptyset$	empty context
Top	Top-type	$\Gamma, x \leq t$	upper bound
$\lambda x \leq t.\ u$	function		
$G(\overline{t})$	module	$\lhd$ ::=	type relations
$: t$ inline$(\overline{s}) = u$	field	$\leq$	subtype
		$\equiv$	equivalence

Notation:

- $\overline{t}$ denotes a sequence of zero or more terms, separated by commas.
- $\overline{d}$ denotes a sequence of zero or more declarations, separated by semicolons.
- dom$(\overline{d})$ denotes the set of labels in the sequence of declarations $\overline{d}$.
- $GT(G_i)$ denotes the module definition named G_i in the global table defined by the program: let $\overline{G} = \overline{M}$ in t.

Evaluation Context:

$$C ::= [] \mid C(t) \mid t(C) \mid C@(t).l \mid t@(C).l \mid C\$ \mid G(\overline{t}, C, \overline{u})$$
$$\mid \lambda x \leq C.\ t \mid \lambda x \leq t.\ C \mid : C \text{ inline}(\overline{s}) = u \mid : t \text{ inline}(\overline{s}, C, \overline{s'}) = u \mid : t \text{ inline}(\overline{s}) = C$$

Reduction: $\boxed{t \longrightarrow t'}$

$$\frac{t \longrightarrow t'}{C[t] \longrightarrow C[t']} \quad \text{(E-Cong)}$$

$$\frac{G(\overline{t}) \rightsquigarrow \mu x \text{ extends } t' \ \{\overline{d}\} \quad O\ l \div u \in \overline{d}}{G(\overline{t})@(s).l \longrightarrow [x \mapsto s]u} \quad \text{(E-Dlg1)}$$

$$\frac{G(\overline{t}) \rightsquigarrow \mu x \text{ extends } t' \ \{\overline{d}\} \quad l \notin \text{dom}(\overline{d})}{G(\overline{t})@(s).l \longrightarrow t'@(s).l} \quad \text{(E-Dlg2)}$$

$$(\lambda x \leq t.\ u)(s) \longrightarrow [x \mapsto s]u \quad \text{(E-App)}$$

$$(: t \text{ inline}(\overline{s}) = u)\$ \longrightarrow u \quad \text{(E-Ext)}$$

Global module lookup: $\boxed{G(\overline{t}) \rightsquigarrow M}$

$$G() \rightsquigarrow GT(G) \quad \text{(E-Lookup1)}$$

$$\frac{G(\overline{t}) \rightsquigarrow \lambda x \leq s.\ M}{G(\overline{t}, u) \rightsquigarrow [x \mapsto u]M} \quad \text{(E-Lookup2)}$$

Figure 3.10: The Deep-- calculus — syntax and operational semantics

The name G is the global identifier for the module, while $(\bar{t})$ are the parameters of the module. Modules may be parameterized by any term, including both types and objects.

By using a global table, there is never any need to compare recursive structures directly. Modules are compared according to their name and parameters: $G(\bar{t}) \equiv G(\bar{u})$ iff $\bar{t} \equiv \bar{u}$. This mechanism is a hybrid between nominal and structural typing; nominal typing is used for the module name itself, while structural typing is used for the parameters [Thorup and Torgersen, 1999].

Nominal subtyping introduces a distinction between programs and terms. A program defines a global module table, while terms are interpreted within the context of an existing module table. Equivalence (or subtyping) between DEEP-- programs is undefined; there is no judgement that would allow us to derive that one program P is equivalent to another program P'. Program equivalence is undefined because it would require a structural comparison between modules. Since all type-checking happens within the context of a single program, program equivalence is only of theoretical interest. It could perhaps be defined using using techniques like meaning preservation [Wells et al., 2003], although we have not attempted to do so.

The use of nominal typing does restrict expressiveness to some extent, because it is not possible to make use of structural record types. In other words, there no way to define a function that accepts arguments of type { foo: Int; }, which denotes any record that has an integer member named foo. All inheritance relationships between modules must be explicitly declared.

However, nominal typing has a practical advantage in terms of efficiency, for exactly the same reason. Modules can be quite large, and a direct structural comparison between large modules is expensive. Nominal typing is significantly faster, which means that it could potentially be used to do run-time type checks, such as those provided by Java's instanceof keyword.

3.5.3 μ-lifting

At first glance, the use of flat global table for modules would seem to run counter to the goals of DEEP--, since it does not allow module definitions to be nested. However, any program which contains nested module definitions can be transformed into one which only has top-level module definitions. We call this process μ-lifting, by analogy with λ-lifting [Johnsson, 1985]. The process closely mimics the mechanism used by current Java compilers, which lift inner classes and anonymous classes to top-level

```
GraphNode = λx ≤ Graph. μy {
  edges: List(x.Edge) = _;
};

GraphEdge = λx ≤ Graph. μx {
  from: x.Node = _;
  to:   x.Node = _;
};

Graph = μx {
  Node:  GraphNode(x);
  Edge:  GraphEdge(x);
  nodes: List(x.Node) = _;
};

ColoredGraphNode = λx ≤ ColoredGraph. μy extends Graph@(x).Node {
  color: Color = _;
};

ColoredGraph = μx extends Graph {
  override Node: ColoredGraphNode(x);
  valid: Bool = and([ n.color != e.to.color |
                      n <- x.nodes, e <- n.edges]);
};
```

Figure 3.11: Example of μ-lifting

---

classes [Gosling et al., 2005] [Igarashi and Pierce, 2000].

Intuitively, the μ-lifting process works because every program has only a finite number of literal module definitions. Each of these definitions is assigned a unique name. It is possible for new modules to be created at run-time when a program is evaluated, but the only run-time operations that DEEP-- supports are function application and delegation, both of which substitute a value for a variable. Any new modules which are created at run-time must therefore be duplicates of ones that were declared in the initial program, with different values substituted for free variables. The parameter list for each module keeps track of these substitutions.

Figure 3.11 shows how the colored graph example given before can be lifted to a set of top-level definitions. Notice that the Node and Edge classes, which were previously nested within Graph, are now parameterized by Graph instead.

Flattening: $\boxed{\lfloor t \rfloor = \mathsf{let}\ \overline{G} = \overline{M}\ \mathsf{in}\ u}$

$$\frac{\begin{array}{c} H \text{ chosen fresh} \\ \overline{x} \leq \overline{s} \vdash \lfloor t \rfloor = \mathsf{let}\ \overline{F} = \overline{M}\ \mathsf{in}\ t' \\ \forall i \in 1..n.\ \ \overline{x} \leq \overline{s},\ y \leq H(\overline{x}) \vdash \lfloor d_i \rfloor = \mathsf{let}\ \overline{G_i} = \overline{N_i}\ \mathsf{in}\ d_i' \end{array}}{\begin{array}{l} \overline{x} \leq \overline{s} \vdash \lfloor \mu y\ \mathsf{extends}\ t\ \{\overline{d}\} \rfloor = \\ \qquad \begin{array}{l} \mathsf{let}\ \overline{F} = \overline{M},\ \overline{G_1} = \overline{N_1},\ ..\ \overline{G_n} = \overline{N_n}, \\ \quad H = \lambda \overline{x} \leq \overline{s}.\ \mu y\ \mathsf{extends}\ t'\ \{\overline{d'}\} \\ \mathsf{in}\ \ H(\overline{x}) \end{array} \end{array}} y \notin \overline{x}$$

$$\frac{\overline{x} \leq \overline{s} \vdash \lfloor t \rfloor = \mathsf{let}\ \overline{G} = \overline{M}\ \mathsf{in}\ t'}{\overline{x} \leq \overline{s} \vdash \lfloor O\ l \div t \rfloor = \mathsf{let}\ \overline{G} = \overline{M}\ \mathsf{in}\ O\ l \div t'}$$

$$\frac{\begin{array}{c} \overline{x} \leq \overline{s} \vdash \lfloor t \rfloor = \mathsf{let}\ \overline{G} = \overline{M}\ \mathsf{in}\ t' \\ \overline{x} \leq \overline{s}, y \leq t' \vdash \lfloor u \rfloor = \mathsf{let}\ \overline{H} = \overline{N}\ \mathsf{in}\ u' \end{array}}{\overline{x} \leq \overline{s} \vdash \lfloor \lambda y \leq t.\ u \rfloor = \mathsf{let}\ \overline{G} = \overline{M},\ \overline{H} = \overline{N}\ \mathsf{in}\ \lambda y \leq t'.\ u'} y \notin \overline{x}$$

$$\frac{\begin{array}{c} \overline{x} \leq \overline{s} \vdash \lfloor t \rfloor = \mathsf{let}\ \overline{F} = \overline{M}\ \mathsf{in}\ t' \\ \overline{x} \leq \overline{s} \vdash \lfloor u \rfloor = \mathsf{let}\ \overline{G} = \overline{N}\ \mathsf{in}\ u' \\ \forall i \in 1..n.\ \ \overline{x} \leq \overline{s} \vdash \lfloor a_i \rfloor = \mathsf{let}\ \overline{H_i} = \overline{P_i}\ \mathsf{in}\ a_i' \end{array}}{\begin{array}{l} \overline{x} \leq \overline{s} \vdash \lfloor : t\ \mathsf{inline}(\overline{a}) = u \rfloor = \\ \qquad \begin{array}{l} \mathsf{let}\ \overline{F} = \overline{M},\ \overline{G} = \overline{N},\ \overline{H_1} = \overline{P_1},\ ..\ \overline{H_n} = \overline{P_n} \\ \mathsf{in}\ \ : t'\ \mathsf{inline}(\overline{a'}) = u' \end{array} \end{array}}$$

$$\overline{x} \leq \overline{s} \vdash \lfloor \mathsf{Top} \rfloor = \mathsf{let\ in\ Top} \qquad \overline{x} \leq \overline{s} \vdash \lfloor x_i \rfloor = \mathsf{let\ in}\ \mathsf{x}_i$$

$$\frac{\begin{array}{c} \overline{x} \leq \overline{s} \vdash \lfloor t \rfloor = \mathsf{let}\ \overline{G} = \overline{M}\ \mathsf{in}\ t' \\ \overline{x} \leq \overline{s} \vdash \lfloor u \rfloor = \mathsf{let}\ \overline{H} = \overline{N}\ \mathsf{in}\ u' \end{array}}{\overline{x} \leq \overline{s} \vdash \lfloor t(u) \rfloor = \mathsf{let}\ \overline{G} = \overline{M},\ \overline{H} = \overline{N}\ \mathsf{in}\ t'(u')}$$

$$\frac{\begin{array}{c} \overline{x} \leq \overline{s} \vdash \lfloor t \rfloor = \mathsf{let}\ \overline{G} = \overline{M}\ \mathsf{in}\ t' \\ \overline{x} \leq \overline{s} \vdash \lfloor u \rfloor = \mathsf{let}\ \overline{H} = \overline{N}\ \mathsf{in}\ u' \end{array}}{\overline{x} \leq \overline{s} \vdash \lfloor t@(u).l \rfloor = \mathsf{let}\ \overline{G} = \overline{M},\ \overline{H} = \overline{N}\ \mathsf{in}\ t'@(u').l}$$

$$\frac{\overline{x} \leq \overline{s} \vdash \lfloor t \rfloor = \mathsf{let}\ \overline{G} = \overline{M}\ \mathsf{in}\ t'}{\overline{x} \leq \overline{s} \vdash \lfloor t\$ \rfloor = \mathsf{let}\ \overline{G} = \overline{M}\ \mathsf{in}\ t'\$}$$

Figure 3.12: Flattening for DEEP--

3.5.3.1 μ-lifting preserves operational semantics

Figure 3.12 defines flattening and unflattening operators. These two operators transform a program from surface to formal syntax, and vice versa.

The flattening operator, written $\lfloor t \rfloor$ will take a term t in the surface syntax for DEEP--, and return a program $\mathsf{let}\ \overline{G} = \overline{M}\ \mathsf{in}\ u$, in the formal syntax of DEEP--. Flattening lifts all nested modules within t to top-level declarations $\overline{M}$, which are assigned to

Unflattening: $\boxed{\lceil t \rceil = u}$

$$\frac{H(\overline{s}) \rightsquigarrow \mu x \text{ extends } t\ \{\overline{d}\}}{\lceil H(\overline{s}) \rceil = \mu x \text{ extends } \lceil t \rceil\ \{\lceil \overline{d} \rceil\}}$$

$$\lceil O\ l \doteqdot t \rceil = O\ l \doteqdot \lceil t \rceil$$

$$\lceil \lambda x \leq t.\ u \rceil = \lambda x \leq \lceil t \rceil.\ \lceil u \rceil$$

$$\lceil : t \text{ inline}(\overline{s}) = u \rceil = (: \lceil t \rceil \text{ inline}(\lceil \overline{s} \rceil) = \lceil u \rceil)$$

$$\lceil \mathsf{Top} \rceil = \mathsf{Top}$$

$$\lceil x \rceil = x$$

$$\lceil t(u) \rceil = \lceil t \rceil (\lceil u \rceil)$$

$$\lceil t@(u).l \rceil = \lceil t \rceil @ (\lceil u \rceil).l$$

$$\lceil t\$ \rceil = \lceil t \rceil \$$$

Figure 3.13: Unflattening for DEEP--

fresh names $\overline{G}$. Flattening is done within a context that provides bounding types for free variables; these bounding types are used to construct the parameter list of each nested module.

Unflattening, written $\lceil t \rceil$ is the opposite of flattening. It takes module references of the form $G(\overline{s})$ in the formal syntax, and rewrites them to modules μx extends $t\ \{\overline{d}\}$ in the surface syntax.

The operational semantics of formal DEEP-- preserves the operational semantics of surface DEEP--. By this we mean the following. Assume we have a program t in the surface syntax, which has been flattened to a program let $\overline{G} = \overline{M}$ in u in the formal syntax. If u reduces to u', then u' will unflatten to a term t', such that t reduces to t'.

In other words, the following reduction strategy is valid for any program: first flatten the program, evaluate the flattened version, and then unflatten it again. We can thus safely confine the type system to flattened programs without loss of generality.

Lemma 3.5.1 (Unflattening is the inverse of flattening) $\lceil \lfloor t \rfloor \rceil = t$.

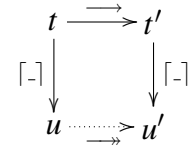
Proof: By induction on t. The base cases are Top and x. All cases except modules follow directly from the induction hypothesis. A module μy extends $t\ \{\overline{d}\}$ is flattened to $H(\overline{x})$, where x are the variables defined in the context where the module occurs. When $H(\overline{x})$ is unflattened, each variable is substituted for itself, thus restoring the original module.

Lemma 3.5.2 (Unflattening preserves substitution)
If $\lceil [x \mapsto s]t \rceil = [x \mapsto \lceil s \rceil] \lceil t \rceil$

Proof: By induction on t.

Lemma 3.5.3 (Unflattening preserves reduction)

Let t be a term in the formal syntax, and u be a term in the surface syntax. If $\lceil t \rceil = u$, and $t \longrightarrow t'$, then $\lceil t' \rceil = u'$ such that $u \longrightarrow\!\!\!\!\!\rightarrow u'$, as illustrated by the following diagram:



Proof: By induction on $t \longrightarrow t'$. Every formal reduction rule has a matching surface reduction rule. β-reduction and delegation follow from lemma 3.5.2. The congruence rules (using $C[t]$) also match, with one exception:

$$\frac{t \longrightarrow t'}{G(\overline{u}, t, \overline{u'}) \longrightarrow G(\overline{u}, t', \overline{u'})}$$

The term $G(\overline{u}, t, \overline{u'})$ unflattens to a module which contains zero or more occurences of $\lceil t \rceil$ as subterms. The term $G(\overline{u}, t', \overline{u'})$ has $\lceil t' \rceil$ in those same positions. By applying the congruence rule zero or more times (along with the induction hypothesis) each occurrence of $\lceil t \rceil$ can be reduced to $\lceil t' \rceil$.

Theorem 3.5.4 (Formal DEEP-- preserves the semantics of surface DEEP--)
If t is a term in the surface syntax, and $\lfloor t \rfloor = \mathsf{let}\ \overline{G} = \overline{M}\ \mathsf{in}\ u$, then $u \longrightarrow\!\!\!\!\!\rightarrow u'$ in the formal syntax implies that $\lceil u' \rceil = t'$, such that $t \longrightarrow\!\!\!\!\!\rightarrow t'$ in the surface syntax.

Proof: By lemmas 3.5.1 and 3.5.3.

3.5.4 Declarative subtyping

The declarative subtyping rules for DEEP-- are shown in Figure 3.14. As in System $\lambda_{\lhd}$, declarative subtyping rules are named according to the pattern DS-*name*. The declarative rules are further subdivided into congruence rules, which compare terms with a similar shape, and "reduction" rules, which do not. The reduction rules are named DS-E*name*. The basic rules can be summarized as follows:

- A module is a subtype of its parent (DS-EINH).
- A field $(: t = u)$ is invariant in t; the implementation u can be overridden. (DS-FIELD1).
- $t@(u).l$ is covariant in t, and invariant in u (DS-DLG).
- $t\$$ is invariant in t (DS-EXT).

Well-subtyping: $\boxed{\Gamma \vdash t \leq_{\text{wf}} u}$

$$\frac{\Gamma \vdash t \text{ wf}, \quad u \text{ wf}, \quad t \lhd u}{\Gamma \vdash t \lhd_{\text{wf}} u} \text{ (W-SUB)}$$

Subtyping & equivalence: $\boxed{\Gamma \vdash t \lhd u}$

$$\frac{\Gamma \vdash s \lhd t, \quad t \lhd u, \quad t \text{ wf}}{\Gamma \vdash s \lhd u} \text{(DS-TRANS)}$$

$$\frac{\Gamma \vdash u \equiv t}{\Gamma \vdash t \equiv u} \text{ (DS-SYM)} \qquad \frac{\Gamma \vdash t \equiv u}{\Gamma \vdash t \leq u} \text{ (DS-EQ)}$$

$$\frac{\begin{array}{c}\Gamma \vdash t \equiv t' \\ \Gamma, x \leq t \vdash u \lhd u'\end{array}}{\Gamma \vdash \lambda x \leq t.\, u \lhd \lambda x \leq t'.\, u'} \text{ (DS-FUN)}$$

$$\frac{G() \rightsquigarrow M}{\Gamma \vdash G() \equiv G()} \text{ (DS-MOD1)}$$

$$\frac{\Gamma \vdash G(\overline{t}) \lhd G(\overline{t'}) \quad u \equiv u'}{\Gamma \vdash G(\overline{t}, u) \lhd G(\overline{t'}, u')} \text{(DS-MOD2)}$$

$$\Gamma \vdash \text{Top} \equiv \text{Top} \qquad \text{(DS-TOP)}$$

$$\frac{x \in \text{dom}(\Gamma)}{\Gamma \vdash x \equiv x} \text{ (DS-VAR)}$$

$$\frac{\Gamma \vdash t \lhd t', \quad u \equiv u'}{\Gamma \vdash t(u) \lhd t'(u')} \text{ (DS-APP)}$$

$$\frac{\Gamma \vdash t \lhd t', \quad u \equiv u'}{\Gamma \vdash t@(u).l \lhd t'@(u').l} \text{ (DS-DLG)}$$

$$\frac{\Gamma \vdash t \equiv u}{\Gamma \vdash t\$ \equiv u\$} \text{ (DS-EXT)}$$

Equivalence (reduction): $\boxed{\Gamma \vdash t \equiv u}$

$$\Gamma \vdash (\lambda x \leq t.\, u)(s) \equiv [x \mapsto s]u \text{ (DS-EAPP)}$$

$$\frac{\begin{array}{c}G(\overline{t}) \rightsquigarrow \mu x \text{ extends } t' \{\overline{d}\} \\ O\, l \doteqdot u \in \overline{d}\end{array}}{\Gamma \vdash G(\overline{t})@(s).l \equiv [x \mapsto s]u} \text{ (DS-EDLG1)}$$

$$\frac{\begin{array}{c}G(\overline{t}) \rightsquigarrow \mu x \text{ extends } t' \{\overline{d}\} \\ l \notin \text{dom}(\overline{d})\end{array}}{\Gamma \vdash G(\overline{t})@(s).l \equiv t'@(s).l} \text{ (DS-EDLG2)}$$

$$\frac{\overline{v} \text{ not empty}}{\Gamma \vdash (:t \text{ inline}(\overline{v}) = u)\$ \equiv u} \text{ (DS-EEXT1)}$$

$$\frac{\begin{array}{c}\Gamma \vdash t \leq_{\text{wf}} G(\overline{t'}) \\ G(\overline{t'}) \rightsquigarrow \mu x \text{ extends } s' \{\overline{d}\} \\ O\, l = u \in \overline{d}\end{array}}{\Gamma \vdash t@(s).l \equiv [x \mapsto s]u} \text{ (DS-EFINAL)}$$

Subtyping (reduction): $\boxed{\Gamma \vdash t \leq u}$

$$\Gamma \vdash t \leq \text{Top} \qquad \text{(DS-ETOP)}$$

$$\frac{x \leq t \in \Gamma}{\Gamma \vdash x \leq t} \text{ (DS-EVAR)}$$

$$\frac{G(\overline{t}) \rightsquigarrow \mu x \text{ extends } u \{\overline{d}\}}{\Gamma \vdash G(\overline{t}) \leq u} \text{ (DS-EINH)}$$

$$\frac{\Gamma \vdash t \leq_{\text{wf}} (:u \text{ inline}(\overline{s}) = u')}{\Gamma \vdash t\$ \leq u} \text{(DS-EEXT2)}$$

$$\frac{\Gamma \vdash t \equiv t'}{\Gamma \vdash (:t \text{ inline}(\overline{s}) = u) \leq (:t' \text{ inline}(\overline{s'}) = u')} \text{ (DS-FIELD1)}$$

$$\frac{\Gamma \vdash t \equiv t', \quad u \equiv u', \quad \overline{s} \equiv \overline{s'}}{\Gamma \vdash (:t \text{ inline}(\overline{s}) = u) \equiv (:t' \text{ inline}(\overline{s'}) = u')} \text{ (DS-FIELD2)}$$

Figure 3.14: Declarative subtyping and type equivalence for DEEP

- If $t \longrightarrow t'$, then $t \equiv t'$ (DS-EDLG1), (DS-EDLG2), (DS-EEXT2). Note that extraction is limited by the inline heuristic, which will be discussed in Section 3.5.4.1.

The most interesting rule is the one for final bindings:

$$\frac{\Gamma \vdash t \leq_{\text{wf}} G(\overline{t'}) \quad G(\overline{t'}) \rightsquigarrow \mu x \text{ extends } s' \{\overline{d}\} \quad O\ l = u \in \overline{d}}{\Gamma \vdash t@(s).l \equiv [x \mapsto s]u}\ (\text{DS-EFINAL})$$

This rule states that if l is final bound in some supertype of t, then we can derive an exact value for $t@(s).l$, even if the value of t itself is not statically known. This capability is important for handling manifest types, such as the animal/cow example given earlier [Leroy, 1994]. Here is the example again, without syntax sugar:

```
...

Cow = μx extends Animal {
  override FoodType     = Grass;
  override favoriteFood = alfalfa;
  eat: λf ≤ x@(x).FoodType. : Animal = ...;
};

feed3: λa ≤ Cow. : Animal = a@(a).eat(hay)$;
```

Within the body of feed3, a.eat is a subtype of $(\lambda f \leq \text{a.FoodType.} : \text{Animal} = \_)$. In order for the application of a.eat to be well-formed, we must show that hay $\leq$ a.FoodType. The value of a is not statically known, so it is not possible to reduce a.FoodType directly. However, using rule (DS-EFINAL), we can derive a@(a).FoodType $\equiv$ Grass because a $\leq$ Cow, and FoodType is final bound within Cow. Since hay inherits from Grass, the body of feed3 is well-formed.

The same technique can be used to implement inlining of final methods, which is a common optimization in object-oriented languages:

```
Matrix = μx {
  rows: Int = _;
  cols: Int = _;
  mdat: Array(Float) = _;

  final getElem(m ≤ Int, n ≤ Int): Float inline(Top) =
    x.mdat[m * x.cols + n];
};

firstVal(mat ≤ Matrix): Float = mat.getElem(0,0);
```

This example implements a matrix, which stores its data in a floating-point array. The overhead of a virtual method call would be unacceptable in a matrix library, since it could easily reduce performance by more than an order of magnitude in matrix-heavy computations. The getElem method is thus declared as final, which is syntax sugar for a final binding, and the field has an inline clause which means "always inline" (see below for explanation). As a result, the call to mat.getElem(0,0) within firstVal can be inlined to mat.mdat
[0∗mat.cols + 0], which an optimizing compiler can further simplify to mat.mdat[0]. The type system guarantees that inlining is safe.

3.5.4.1 Inlining, fields, and separate compilation

The inline clause in a field is used to control when inlining takes place. Assume f is a field which is statically known, e.g. $f \equiv (: t \ \mathsf{inline}(\overline{s}) = u)$, and we want to find the principal supertype of $f\$$. A type checker has two options that it can take. It may choose to extract the implementation of f using rule (DS-EExt1), yielding the judgement $f\$ \equiv u$. Alternately, it may choose to extract the interface of f using rule (DS-EExt2), which yields the judgement $f\$ \leq t$.

The inline clause is a heuristic which guides the type-checker in making this decision. If the inline clause is empty (i.e. $\overline{s}$ is an empty sequence), then inlining is not allowed, and the type-checker is forced to generalize to the interface. If the inline clause contains expressions which can all be reduced to values, then inlining is allowed.

In the matrix example above, inline(Top) means "always inline", because Top is a value. Fields with an empty inline clause, such as those in the previous examples in this chapter, mean "never inline". More complex heuristics (meaning "inline under certain conditions") are used to ensure that type-checking is decidable in the presence of general recursion, and will be discussed in Section 3.6.

3.5.5 Well-formedness

The well-formedness rules for DEEP-- are shown in Figure 3.15. Well-formedness over terms makes the following static type checks:

- In a function call $t(u)$, it ensures that t is a function, and that u is a subtype of the input type for that function (W-APP).
- For delegation $t@(u).l$, it ensures that t is a module which defines a slot l, and that u is a subtype of t (W-DLG).

Context well-formedness: $\boxed{\Gamma \text{ wf}}$

$$\emptyset \text{ wf} \qquad \text{(W-GAM1)}$$

$$\frac{\Gamma \text{ wf}, \quad x \notin \text{dom}(\Gamma) \quad \Gamma \vdash t \text{ wf}}{\Gamma, x \leq t \text{ wf}} \qquad \text{(W-GAM2)}$$

Term well-formedness: $\boxed{t \text{ wf}}$

$$\frac{\Gamma \text{ wf} \quad x \in \text{dom}(\Gamma)}{\Gamma \vdash x \text{ wf}} \qquad \text{(W-VAR)}$$

$$\frac{\Gamma \text{ wf}}{\Gamma \vdash \mathsf{Top} \text{ wf}} \qquad \text{(W-TOP)}$$

$$\frac{\Gamma, x \leq t \vdash u \text{ wf}}{\Gamma \vdash \lambda x \leq t.\, u \text{ wf}} \qquad \text{(W-FUN)}$$

$$\frac{G() \rightsquigarrow M}{\Gamma \vdash G() \text{ wf}} \qquad \text{(W-MOD1)}$$

$$\frac{\Gamma \vdash G(\overline{t}) \text{ wf} \quad G(\overline{t}) \rightsquigarrow \lambda x \leq s.\, M \quad \Gamma \vdash u \leq_{\text{wf}} s}{\Gamma \vdash G(\overline{t}, u) \text{ wf}} \qquad \text{(W-MOD2)}$$

$$\frac{\Gamma \vdash \overline{s} \text{ wf}, \quad u \leq_{\text{wf}} t}{\Gamma \vdash (: t\ \mathsf{inline}(\overline{s}) = u) \text{ wf}} \qquad \text{(W-FIELD)}$$

$$\frac{\Gamma \vdash t \leq_{\text{wf}} (\lambda x \leq s.\, \mathsf{Top}) \quad \Gamma \vdash u \leq_{\text{wf}} s}{\Gamma \vdash t(u) \text{ wf}} \qquad \text{(W-APP)}$$

$$\frac{\Gamma \vdash t \leq_{\text{wf}} G(\overline{s}) \quad G(\overline{s}) \rightsquigarrow \mu x\ \mathsf{extends}\ t'\ \{\overline{d}\} \quad O\ l \doteqdot u' \in \overline{d} \quad \Gamma \vdash u \leq_{\text{wf}} t}{\Gamma \vdash t@(u).l \text{ wf}} \qquad \text{(W-DLG)}$$

$$\frac{\Gamma \vdash t \leq_{\text{wf}} (: u\ \mathsf{inline}(\overline{s}) = u')}{\Gamma \vdash t\$ \text{ wf}} \qquad \text{(W-EXT)}$$

Program well-formedness:

$$\frac{\overline{G} :: \emptyset \vdash \overline{M} \text{ wf} \quad \emptyset \vdash t \text{ wf} \quad \text{There are no illegal cycles.}}{\mathsf{let}\ \overline{G} = \overline{M}\ \mathsf{in}\ t \text{ wf}} \qquad \text{(W-PROG)}$$

Module wf: $\boxed{G(\overline{s}) :: \Gamma \vdash M \text{ wf}}$

$$\frac{G(\overline{s}, x) :: \Gamma, x \leq t \vdash M \text{ wf}}{G(\overline{s}) :: \Gamma \vdash \lambda x \leq t.\, M \text{ wf}} \qquad \text{(W-MFUN)}$$

$$\frac{\Gamma \vdash t \text{ wf} \quad \Gamma, x \leq G(\overline{s}) \vdash \overline{d} \text{ wf} \quad \overline{d} \text{ has no duplicate labels.}}{G(\overline{s}) :: \Gamma \vdash \mu x\ \mathsf{extends}\ t\ \{\overline{d}\} \text{ wf}} \qquad \text{(W-MDEF)}$$

Declaration well-formedness: $\boxed{\Gamma \vdash d \text{ wf}}$

$$\frac{\Gamma, x \leq G(\overline{s}) \vdash t \text{ wf} \quad G(\overline{s}) \rightsquigarrow \mu y\ \mathsf{extends}\ u\ \{\overline{d}\} \quad \Gamma \vdash u.l \text{ undefined}}{\Gamma, x \leq G(\overline{s}) \vdash \mathsf{def}\ l \doteqdot t \text{ wf}} \qquad \text{(W-DECL)}$$

$$\frac{\Gamma, x \leq G(\overline{s}) \vdash t \leq_{\text{wf}} u@(x).l \quad G(\overline{s}) \rightsquigarrow \mu y\ \mathsf{extends}\ u\ \{\overline{d}\} \quad \Gamma \vdash u.l \text{ virtual}}{\Gamma, x \leq G(\overline{s}) \vdash \mathsf{override}\ l \doteqdot t \text{ wf}} \qquad \text{(W-ODECL)}$$

virtual/undefined:

$$\Gamma \vdash \mathsf{Top}.l \text{ virtual} \qquad \Gamma \vdash \mathsf{Top}.l \text{ undefined}$$

$$\frac{\Gamma \vdash t \equiv G(\overline{s}) \quad G(\overline{s}) \rightsquigarrow \mu x\ \mathsf{extends}\ t'\ \{\overline{d}\} \quad O\ l = u \notin \overline{d} \quad \Gamma \vdash t'.l \text{ virtual}}{\Gamma \vdash t.l \text{ virtual}}$$

$$\frac{\Gamma \vdash t \equiv G(\overline{s}) \quad G(\overline{s}) \rightsquigarrow \mu x\ \mathsf{extends}\ t'\ \{\overline{d}\} \quad l \notin \text{dom}(\overline{d}) \quad \Gamma \vdash t'.l \text{ undefined}}{\Gamma \vdash t.l \text{ undefined}}$$

Figure 3.15: Well-formedness rules for DEEP--

- For extraction $t\$$, it ensures that t is a field (W-EXT).
- For a module $G(\bar{s})$, it ensures that the parameters $\bar{s}$ have the appropriate types (W-MOD).
- For a field $(:t = u)$, it ensures that u is a subtype of t (W-FIELD).

Well-formedness is also defined over programs. A program is well-formed if all of its modules are well-formed, and if there are no "illegal cycles". We will discuss illegal cycles in the next section, and present an algorithm for detecting them.

A module definition is well-formed if every slot is well-formed, and if the following three conditions hold:

- For every non-overriding slot $(\mathsf{def}\ l \doteqdot u)$, the slot l cannot be defined in any of the parents of the module. This check is done by $t.l$ undefined, where t is the parent of the module.
- For every overriding slot $(\mathsf{override}\ l \doteqdot u)$, the slot l must be defined, but not final bound, in all parents. This check is done by $t.l$ virtual.
- For every overriding slot $(\mathsf{override}\ l \doteqdot u)$, the term u must be a subtype of $t@(x).l$, which denotes the slot named l in the parent.

Because of the way in which well-formedness is defined, a module definition is only well formed if the parent t is statically known. A term t is *statically known* if it possible to derive $t \equiv v$ for some value v.

There are two reasons why t must be statically known. First, the judgements $t.l$ virtual, and $t.l$ undefined, must be able to traverse the entire list of parents. This traversal ensures that no two slots have the same name, and that final bindings are not overridden. Second, for an overriding slot $(\mathsf{override}\ l \doteqdot u)$, u must be a subtype of $t@(x).l$ in order for inheritance to be sensible; a derived module must specialize definitions in the base module. If t cannot be reduced to a module, then $t@(x).l$ has only trivial subtypes.

This restriction on inheritance (the fact that the parent of a module must be statically known) is standard within most statically-typed OO languages, including Java, C++, and even Beta. Nevertheless, we regard it as a crippling limitation of DEEP--. It is not possible to inherit from a variable, or to inherit from a virtual class. Since inheritance and instantiation are the same thing, it is not possible to instantiate a virtual class either. There are ways of working around this problem in some cases, but in general, virtual classes are much less useful than they might otherwise be.

This limitation on inheritance is the primary difference between the DEEP-- and DEEP calculi. The DEEP calculus, which is presented in Chapter 4, lifts the limitation on inheritance, and thus allows us to solve the DSL expression problem.

3.6 Partial Evaluation and decidability

In a traditional type system, general recursion operates only at the level of objects; recursion at the type level is either non-existant, or tightly controlled. By restricting recursion at the level of types, and maintaining a clear separation between types and objects, it is possible for typing to be decidable even if the language of objects is not strongly normalizing.

Subtyping in DEEP-- is not decidable. General recursion in DEEP-- is defined over all terms, and no restrictions are placed on its use. Subtyping in System $\lambda_{\triangleleft}$ is also undecidable, but undecidability in System $\lambda_{\triangleleft}$ stems from Girard's paradox, which is very difficult to trigger in practice. In contrast, undecidability in DEEP-- is very easy to trigger, as will be shown in the following examples.

The design philosophy of DEEP-- is to accept undecidability as a necessary evil, but to make it manageable in practice. Almost every piece of software currently in use is written in a language that supports general recursion, and software engineers understand the reasons why programs sometimes hang. Instead of restricting the type system so that the compiler is guaranteed to terminate, our goal is to ensure that non-termination is predictable.

In other words, the algorithms used for type-checking should not violate programmer's expectations about when recursion is safe. There should be simple guidelines for writing programs in such a way that type-checking terminates. Obvious violations of those guidelines should be detected and trapped as errors. Less obvious violations may cause the compiler to hang, but it should still hang for a reason that is either obvious to the programmer, or can be located using an interpreter or debugger, and which can be corrected by modifying the program.

We hypothesize that by following these guidelines, recursion can be made sufficiently well-behaved that lack of decidability will not be an issue in practical programming. Of course, there is no way to test this hypothesis short of implementing DEEP, and using it to construct real-world programs.

3.6.1 Breaking illegal cycles

The following definition will cause the proof of well-formedness to diverge:

```
M = μx {
  infinity = x.infinity + 1;
}
```

In order to prove that infinity + 1 is well-formed, the type system must ensure that infinity is a subtype of Int. The subtype comparison will attempt to evaluate x.infinity, a computation that does not terminate. We will refer to definitions such as the one above as *illegal cycles*. Such cycles are not well-formed, because they have no finite proof of well-formedness.

Programming Guideline #1: hide recursive definitions behind fields. An alternative definition of infinity that is well-formed is:

```
M = μx {
  infinity: Int = x.infinity + 1;
}
```

In this definition, infinity is defined using a field. The field breaks the illegal cycle by first declaring infinity to be a subtype of Int, and then proving that it is actually a subtype of Int within the context of that declaration. In general, all recursive function definitions must be hidden behind fields in this way.

Note that this mechanism for breaking illegal cycles is identical to the way in which the fix operator is used to define fixpoints in other languages. In the following example, y is first assumed to have type Int, and the body of the function is then type-checked under that assumption:

```
infinity = fix λy: Int. y+1
```

Recursive data structures and type definitions do not need to be hidden behind fields:

```
M = μx {
  Stream(T ≤ Top) = μy {
    head: T;
    tail: x.Stream(T);
  };

  ones = μy extends x.Stream(Int) {
    override head = 1;
    override tail = x.ones;
  };
}
```

The difference between this definition and the previous one is that the recursive paths x.Stream(T) and x.ones are *contractive* — they terminate immediately to yield a value [Pierce, 2002]. The well-formedness proof must demonstrate that x.ones is a subtype of x.Stream(Int), but this result follows immediately from the definition of ones.

3.6.2 Detecting illegal cycles with lazy type checking

In practice, it is relatively easy to detect and trap most illegal cycles by using a lazy algorithm. Every term in a program is colored one of three colors: black, white, or red. Initially, all terms in the program are black, which means that they have not yet been proven to be well-formed. When the well-formedness proof starts working on a particular term, it first colors the term red, indicating that the well-formedness proof is in progress. If the proof succeeds, it colors the term white, indicating that the term is well-formed.

When the well-formedness proof encounters a recursive path, such as x.ones, it will check the color of the target term that the path points to before it expands the path. If the target term is white, then the path can be safely expanded. If the target term is black, then it must be proven well-formed before it can be used. If the term is red, then an illegal cycle has been detected, and the type-checker will indicate an error.

The algorithm is lazy because of the way it handles values. A function will be colored white as soon as the type of its argument is white. A module is white as soon as its parent is white, and a field is white as soon as its interface is white.

Function bodies, module slots, and field implementations are all checked lazily. Once the argument type of a function f has been verified, it is possible to determine whether calls to that function (e.g. $f(a)$) are well-formed. Unlike a traditional type system, the well-formedness judgement does not calculate the type of $f(a)$ immediately. The body of f is only needed when $f(a)$ is actually reduced, so type-checking of the body can be deferred until that time. The same holds for slots and field implementations. A slot will not be checked until it is projected from a module, and an implementation will not be checked until it is extracted from the field.

The term coloring algorithm presented here corresponds exactly to the way lazy evaluation is done in functional languages like Haskell [Marlow and Peyton Jones, 1998] [Peyton Jones, 1992]. A black term is essentially a *thunk*; an unevaluated piece of code. *Forcing* a thunk will cause its value to be computed, and is done only when that value is requested by some other computation. A white term is a thunk that has

already been forced. A red term is a *black hole*. When a thunk is forced, it is temporarily overwritten with a black-hole. Any attempt to force a black hole will generate an illegal-cycle error.

3.6.3 Inlining and partial evaluation

A field of the form $(: t = u)$ (in which the inline clause is empty) completely hides the implementation u behind the interface t. Implementation hiding is often a good thing. Not only does it allow recursive definitions, it allows separate compilation, and it's an important part of good software design. Implementation hiding makes it possible to change u without affecting the rest of the program.

Nevertheless, it is sometimes necessary to expose u in certain circumstances. We refer to such "implementation exposure" as *inlining*, because it closely resembles the inlining optimization which is widely used in compilers. Inlining in DEEP-- is not just an optimization, however. Inlining is built into the subtype relation, and it is an integral part of the type system. In particular, the DEEP-- calculus uses inlining and partial evaluation to deal with dependent types.

Although inlining is widely used in modern compilers, it does pose some challenges. On the one hand, inlining is easy to implement and can be very effective. On the other hand, deciding when and where to inline can be extremely difficult. The biggest potential gains, and the most difficult challenges, arise when inlining is combined with recursion. Most compilers will not inline recursive functions, because in the presence of recursion, inlining turns into full-blown *partial evaluation* [Jones et al., 1993].

For every function application, a partial evaluator has a choice of whether to specialize (i.e. inline) the application, or whether to *residualize* the application, in which case it will be left as-is in the compiled code. Choosing to specialize will increase program size, and may lead to an an infinite chain of specializations if the function is recursive. As a result, the evaluator may or may not terminate, depending on what strategy it uses to make this choice. Aggressive strategies can do more optimizations, but they are also more likely to hang. Even if they do terminate, they may produce "bloated code" that reduces performance instead of improving it. A conservative strategy yields smaller code and is more likely to terminate, but may not take advantage of important inlining opportunties.

A wide variety of techniques have been proposed in the partial evaluation litera-

ture to help guide the evaluator, such as staging and binding-time analysis [Jones and Glenstrup, 2002] [Taha, 2003]. Section 5.8.1 discusses these alternatives in more detail. The DEEP-- calculus is much simpler; it does not attempt to do automatic analysis of any kind. Instead, it uses an old technique which is similar to the *filters* found in Schism [Consel, 1993].

Programming Guideline #2: list all arguments that control recursion in the inline clause. Every recursive function uses one or more of its arguments to control recursion. If these arguments are statically known for some particular application of that function, then inlining is likely to terminate, and will probably be of benefit. A term t is said to be "statically known" if it is possible to prove that $t \equiv v$ for some value v.

The use of the inline clause can be illustrated by the simple power function, which calculates x^n:

```
M = μm {
  final pow(x ≤ Int, n ≤ Int): Int inline(n) =
    if (n == 0) then 1 else x * m.pow(x, n−1);

  square(y: Int): Int = m.pow(y,2);   // inlined
  exp2  (y: Int): Int = m.pow(2,y);   // not inlined
};
```

In the case of pow, the "recursion argument" is n. If the argument passed for n is an integer value, or can be reduced to one, then the if-then-else expression can be eliminated, and recursion will terminate. If the argument for n is a variable or an expression with an unknown value, then inlining is not only a waste of time, but potentially dangerous; it could cause the partial evaluator to hang. The inline(n) clause explicitly declares this property.

The application pow(y,2) within the body of square can be safely inlined, because 2 is a value. It is thus possible to derive m.pow(y,2) ≡ y*y*1. The application pow(2,y) cannot be safely inlined, because the value of y is not statically known. DEEP-- will treat pow(2,y) as a residual which cannot be further reduced. The application pow(x,n-1) within the body of pow is also protected from inlining for the same reason.

3.6.4 Verifying inline directives

If the compiler fails to terminate, the most likely reason is an incorrect inline clause. DEEP-- has no way to verify that the programmer has specified the correct variables for safe inlining; it must accept on faith that the directive is an accurate guide.

The lazy coloring algorithm for type-checking that was described earlier will locate

some of the more obvious misuses of the inline clause, but not all of them. If pow was declared with inline(Top), which means "always inline", then that mistake would be flagged as an illegal cycle, because the compiler would attempt to inline pow(x,n-1) before the body of pow was proven well-formed.

However, if pow was declared with inline(x), which specifies the wrong argument, then it would not be flagged as an error. Instead, the compiler would hang when it encountered pow(2,y), because inlining would descend into an infinite series of nested if-then-else expressions.

3.6.5 Complex inline directives

The inline clause can contain more than simple variables. The following example shows the pow function again, but this time it can only be inlined if the argument n is less than or equal to 10. By putting a cap on the maximum value of n, this example limits the depth of recursion and the amount of code that will be generated at compile-time.

```
M = μm {
  dynFalse: Bool = false;
  final dynTest(b: Bool): Bool inline(b) =
    if (b) then true else m.dynFalse;

  final pow(x ≤ Int, n ≤ Int): Int inline(m.dynTest(n<=10)) =
    if (n == 0) then 1 else x * m.pow(x, n-1);

  kilobyte = m.pow(2,10);      // inlined to 1024
  megabyte = m.pow(2,20);      // not inlined
};
```

If n $\leq$ 10, then dynTest will return true, which is a value. If n $>$ 10, then dynTest will return m.dynFalse. The implementation of dynFalse is fully hidden behind a field, so m.dynFalse is a residual expression, the value of which is not statically known. If n $\leq$ 10 is unknown, then m.dynTest(n $\leq$ 10) is also unknown.

3.6.6 Dependent types and partial evaluation

Partial evaluation is integrated into the type system of DEEP-- because it is necessary for handling full dependent types. A classic example of a dependently-typed function is the sprintf function in C. This function takes a formatting string, followed by some number of arguments with arbitrary types. The formatting string determines the number and type of arguments which are expected, e.g.

```
final splitFmt(str ≤ String, m ≤ Int, n ≤ Int): List(String)
              inline(str,m,n) =
  if (n >= str.length) then Nil
  else if (str.charAt(n) == '%') then
    if (n > m)
      Cons(str.substring(m,n),   splitFmt(str, n, n))
    else if (n+2 <= str.length)
      Cons(str.substring(n,n+2), splitFmt(str, n+2, n+2))
    else Cons("%", Nil)
  else splitFmt(str, m, n+1);

final processFmt(cmd ≤ String, ins ≤ List(String),
                 out ≤ String): Top inline(cmd) =
  if (cmd == "%d") then
    λ(a ≤ Int). sprintfTail(ins, out.concat(int2String(a)))
  else if (cmd == "%s") then
    λ(a ≤ String). sprintfTail(ins, out.concat(a))
  else sprintfTail(ins, out.concat(cmd));

final sprintfTail(ins ≤ List(String), out ≤ String): Top
                  inline(ins) =
  case ins of {
    Nil        -> (: String = out)
    Cons(x,xs) -> processFmt(x, xs, out)
  }

sprintf(cmdStr: String) = sprintfTail(splitFmt(cmdStr,0,0), "");
```

Figure 3.16: The sprintf function.

```
sprintf("%d bottles of %s on the wall.", 99, "beer");
sprintf("%s has found %d %s so far.", "Bopeep", numSheep, "sheep");
```

In C, this function is a blatant breach of type safety, but it can be made statically safe using dependent types [Augustsson, 1998]. Figure 3.16 shows the code for sprintf in DEEP--. This code is not as elegant as it could be, because we are trying to balance clarity of presentation with the need to keep close to the core calculus. We assume that the standard operations found in the Java String class are defined on strings.

The splitFmt function splits the formatting string into a list, e.g.

```
splitFmt("%d bottles of %s on the wall.") ⟶
  ["%d", " bottles of ", "%s", " on the wall."]
```

The sprintfTail function then recurses over the list. For every formatting command of the form "%d" or "%s", it will return a function which accepts an argument of the

appropriate type, converts the argument to a string, and appends it to the output string. When it reaches the end of the list, sprintfTail returns a field.

If the first argument to sprintf is a constant string, DEEP-- will use partial evaluation to generate a function which accepts the correct number of additional arguments, with the correct types. If the argument *s* is not statically known, then then the static type of sprintf(*s*) will be Top, which effectively prevents the expression from being used in any computations.

The DEEP-- version of sprintf can be compared to solutions in two other languages. Augustsson presents a solution for Cayenne, which is a dependently typed version of Haskell, while Sheard and Peyton Jones present a solution for Template Haskell [Augustsson, 1998] [Sheard and Peyton Jones, 2002].

Cayenne does not do inlining or code generation; it merely allows sprintf to be statically typed. The Cayenne solution splits sprintf into two recursive functions: one which calculates the type, and one which actually converts the arguments to a string. Only the type is calculated at compile-time; all computation with objects is delayed until run-time.

Template Haskell does not support dependent types; it relies on code generation instead. There is only one recursive function. A call to sprintf will first generate code at compile-time, and then type-check the generated code. In other words, type-checking happens after code generation, instead of before code generation. Template Haskell achieves many of the benefits of dependent types within a much simpler type system by switching back and forth between type-checking passes and code generation passes.

The DEEP-- solution blends certain aspects of both of these techniques. Like Cayenne, DEEP-- is dependently typed. Type checking of all functions is done when they are declared, before any code generation takes place. Like Template Haskell, however, DEEP-- will generate code at compile-time. The DEEP-- solution uses only one recursive function, although it is split here into two mutually recursive parts for presentation purposes. Moreover, in DEEP-- the function is written in a "generative style"; it is very similar to the Template Haskell version.

Because Cayenne distinguishes between types and objects, it must calculate the type of sprintf("...") separately. Template Haskell also calculates the type separately, although it does so by generating code, and then assigning a type to the generated code. In DEEP--, the generated code *is* the type.

An object expression serves as a singleton type for the purpose of type-checking. A singleton type, by definition, is the most accurate type which can be calculated in

any given situation. In DEEP--, the "type" of `1 + 2` is not `Int`. The "type" of `1 + 2` is `3`. Less specific types, such as `Int`, are used only when the type-checker is forced to generalize, usually because the precise implementation is hidden behind a field.

3.6.7 Conclusion

Dependent types, subtyping, and partial evaluation are often regarded as separate techniques. Most of the research on partial evaluation has been done for dynmically-typed languages like Scheme, so the interaction between partial evaluation and static typing has not been well-studied. Most dependent type systems do not support subtyping, and they are seldom used for code generation. Most object-oriented languages make heavy use of subtyping, but seldom support either code generation or full dependent types.

DEEP-- demonstrates that these three mechanisms can be viewed as different facets of the same technique, and can be smoothly integrated within the same formal system. The idea of using explicit `inline` guards comes directly from the literature on partial evaluation, but it is used in DEEP-- to help ensure that type-checking terminates when general recursion is combined with dependent types.

3.7 Limitations

The previous sections provided several examples of well-known problems that DEEP-- can handle sucessfully. DEEP-- provides a module system which supports virtual classes, it defines inheritance over modules, and inheritance can be used to extend virtual class families. DEEP-- also provides a mechanism for performing type-safe partial evaluation and compile-time code generation.

However, there are a number of problems that DEEP-- cannot handle, and they are just as important. This section covers these limitations in detail. As with any formal system, there is always room for improvement. The main limitations are:

1. It is not possible to inherit from a virtual class, or even to instantiate one.
2. There is no support for multiple inheritance, or even Java-like interfaces.
3. There is no support for abstract classes.
4. There are no wildcards or existential types.

The first two limitations in this list are resolved by DEEP, and are the subject of the next chapter. The second two limitations are not resolved by DEEP, but present no

fundamental difficulties; we have ignored them purely in the interest of simplicity.

3.7.1 It is not possible to extend a virtual class.

Although DEEP-- allows virtual classes to be declared, and it allows modules that contain virtual classes to be extended, it is not possible to extend a virtual class directly. Extending a virtual class is illegal because the parent of a module must be statically known in order to prove that the module is well-formed. In other words, inheritance and instantiation are both illegal for virtual classes; a class must be final bound before it can be used.

3.7.1.1 Factories

```
AbstractPointMod = μm {
  Point: μp {
    x: Int = _;
    y: Int = _;
  };
  makePoint(ix: Int, iy: Int): m.Point = _;

  someGeometry = ... makePoint(0,0) ...;
};

ConcretePointMod = μm extends AbstractPointMod {
  override Point = μp extends super_m.Point;  // Point is final

  override makePoint(ix: Int, iy: Int): m.Point =
    μp extends Point {
      override x: Int = ix;
      override y: Int = iy;
    };
};
```

Figure 3.17: The factory design pattern

The restriction on instantiation can be circumvented in part by using the factory design pattern [Gamma et al., 1995], as shown in Figure 3.17. The code declares a module named AbstractPointMod with a virtual class Point, and a factory method makePoint. When AbstractPointMod is later extended, the Point class is finalized — it changes from a virtual class to a class which is final bound. Once Point has been final bound, it possible to specify an implementation for makePoint.

The advantage of the factory method design pattern is that it is a well-engineered abstraction. Client code only needs to know the name of a class (e.g. Point), and the name of a function which creates instances of that class (e.g. makePoint). Client code does not need to know exactly how the class is implemented, or how it is constructed; those details have been abstracted.

The disadvantage of the factory method design pattern is that AbstractPointMod is useless on its own. It is necessary to create a concrete version, even though that version does not add any real content to the module. Putting the implementation of makePoint inside AbstractPointMod would be more convenient, and it would not harm abstraction; subsequent modules could still override it.

3.7.1.2 Higher order hierarchies

The inability to inherit from a virtual class is more than an inconvenience, it is a crippling blow, because there is no effective workaround. Although there are some class families that do not use inheritance, such as the Graph family presented earlier, they are in the minority. The majority of object-oriented class libraries, even small libraries, are frameworks with inheritance relationships between classes. If such frameworks were encoded within DEEP--, all base classes would have to be either top-level classes, or final bound, a situation that is scarcely better than what mainstream OO languages already offer.

Some experimental languages, such as gbeta, allow inheritance from a virtual class, and thus can be used to implement *higher order hierarchies* [Ernst, 1999b] [Ernst, 2003]. Like gbeta, DEEP removes the restriction on inheritance. Modules and inheritance in DEEP are truly scalable; they can be used to extend class frameworks of arbitrary size and complexity.

3.7.2 There is no support for multiple inheritance.

Multiple inheritance has always been a difficult problem for object-oriented languages. On the one hand, the need for multiple inheritance seems to arise quite often, in many different classification hierarchies. For example, a dolphin is both a mammal, and an animal that swims. It is thus natural to assume that it would inherit characteristics of both mammals (warm blood etc.) and swimmers (tail, fins, etc.). As the example in Section 3.4.5 regarding Haskell type classes illustrates, multiple inheritance is sometimes required even in situations where one would not expect to need it.

On the other hand, it has proved quite difficult to come up with a satisfactory mechanism for implementing multiple inheritance in practice. There is no magic solution, and every OO language handles the problem somewhat differently.

Multiple inheritance is the subject of the next chapter. The same technique that DEEP uses to implement multiple inheritance can also be used to allow inheritance from a virtual class. Although these two problems appear different, they have the same solution.

3.7.3 There is limited support for abstract classes

Many OO languages, including both Java and C++, allow *abstract classes*. An abstract class does not provide implementations for all of its methods; an unimplemented method is called an *abstract method*.

There are two reasons why an implementation might not be provided. First, abstract classes often sit at the top of the inheritance hierarchy, and there may simply not be enough information at the point where the class is declared to supply a default implementation. For example:

```
Stream(T ≤ Top) = μx {
  head: T          = _;
  tail: Stream(T)  = _;
};
```

The type of head is T, which is a type parameter. There is no way of constructing a "default" value for a completely unspecified type.

Second, a base class often provides the skeleton of an algorithm which may be instantiated to several different versions, following the template method design pattern [Gamma et al., 1995]. Such classes work much like higher-order functions in functional languages. Each derived class "fills in the blanks" by providing the appropriate implementations.

Java and C++ will check to make sure that a derived class actually does provide all of the implementations that it is supposed to; their type systems ensure that all "blanks" have been filled in. DEEP-- does not make this check.

DEEP-- fakes abstract classes by allowing the interface of a field to be used as a placeholder for the implementation. The above example is syntax sugar for:

```
Stream(T ≤ Top) = μx {
  head: T          = T;
  tail: Stream(T)  = Stream(T);
};
```

Whether or not this definition is meaningful is open to debate. Stream(Int) is intended to be used as a type, but it can also be interpreted as an object. As an object, it is a circular data structure that represents the infinite sequence: Int, Int, Int, Int... As far as DEEP-- is concerned, this data structure can be legally used wherever a stream of integers is expected.

DEEP-- does not check to make sure that abstract methods have been implemented because as far as it is concerned, the methods have been implemented. Calling an "unimplemented" method at run-time will not generate an error, it will merely return the interface of the field.

Although DEEP-- is type-safe in the technical sense (or at least we believe it to be so), it has omitted a form of program verification that is useful in practice. The users of an abstract class should be informed if they fail to implement all of the methods that they are supposed to.

The solution to this problem is to distinguish between abstract prototypes, which may have missing implementations, and concrete prototypes, which are fully defined. The universe judgement that we described in Section 2.3.4 is well-suited to making this distinction. Abstract prototypes would live in the universe of types, while concrete prototypes would live in the universe of objects.

3.7.4 There is no support for wildcards

Parametric polymorphism doesn't always interact with subtype polymorphism as well as one might hope. For example, although one might expect both Vector(Int) and Vector(Float) to be subtypes of Vector(Number), they are actually two unrelated types. The argument to a type constructor must be invariant for reasons of type safety.

Java 1.5 introduced *wildcards*, which are an excellent solution to this problem. [Torgersen et al., 2004]. Wildcards allow types such as Vector(? $\leq$ Number). The ? symbol matches any type, so Vector(? $\leq$ Number) is a supertype of both Vector(Float) and Vector(Int).

The same technique is potentially useful for type families. In the graph example given earlier, ColoredGraph.Node is not a subtype of Graph.Node. However, ColoredGraph.Node would be a subtype of (? $\leq$ Graph).Node. There are times when it is important to know the exact family that a class belongs to, and there are times when it doesn't matter. Wildcards are a convenient way of dealing with the "don't care" cases. The type (? $\leq$ Graph).Node is a supertype of the *Node* class in every graph family.

Neither DEEP-- nor DEEP support wildcards, although modules can be used to build existential types that mimick wildcards to some extent. In the example below, SomeGraphNode(ColoredGraph, ColoredGraph.Node) is a subtype of AnyGraphNode.

```
AnyGraphNode = μx {
  G:    Graph;
  Node: G.Node;
};

SomeGraphNode(Gr ≤ Graph, Nd ≤ Gr.Node) = μx extends AnyGraphNode {
  override G    = Gr;
  override Node = Nd;
};
```

Modules are not ideal in this situation for two reasons. First, subtyping between modules is nominal, rather than structural, so they are less convenient for cases like this; AnyGraphNode must be declared as a named type. Second, the programmer must pack and unpack the existential by hand. Wildcards are convenient precisely because they can be declared on the fly, and they are packed and unpacked automatically.

There is no technical reason why proper wildcards could not be added to DEEP--; they have been omitted purely in the interest of keeping the core calculus simple.

Chapter 4

Mixins and Multiple Inheritance

Combine all ingredients in a bowl, and stir until thoroughly mixed.

— a common instruction in recipes

4.1 Introduction

Multiple inheritance has always been a challenging problem in object-oriented languages. On the one hand, it seems like a natural and intuitive extension of single inheritance. On the other hand, multiple inheritance has proved difficult to implement in practice; there are almost as many different mechanisms for multiple inheritance as there are object-oriented languages. Moreover, every mechanism offers different advantages and disadvantages, with no accepted "best" solution.

The DEEP-- calculus presented in Chapter 3 does not support multiple inheritance of any kind. It also has another major limitation: it does not support *virtual inheritance*, which is inheritance from a virtual class. This chapter introduces the DEEP calculus, which eliminates both of these restrictions.

Although multiple inheritance and virtual inheritance appear quite different, they are really two facets of the same problem. In DEEP--, the parent of a module must be statically known, a restriction that DEEP-- shares with most other OO languages. The DEEP calculus eliminates this restriction; it allows the parent of a module to be any term. By eliminating the restriction on inheritance, DEEP is able to support both virtual inheritance, and a form of multiple inheritance called *mixins*. Both cases are illustrated below, using code that would be illegal in DEEP--, but is legal in DEEP.

$M = \mu x \ \{$

```
  A: μy { ... };
  B: μy extends x.A { ... };  // inheriting from a virtual class
};

C = μy { ... };
D = λsuper ≤ C. μy extends super { ... }; // mixins
E = λsuper ≤ C. μy extends super { ... };
```

In the first case, M.B inherits from M.A, where M.A is a virtual class. In the second case, D and E are both defined as functions that extend their arguments. In essence, D and E are classes that can be "mixed" together in various combinations; possible combinations in this case are D(C), E(C), E(D(C)), and D(E(C)). As we shall demonstrate in this chapter, mixins have several nice properties that make them a particularly good way of implementing multiple inheritance.

At the end of this chapter, we will be able to present solutions to both the expression problem, and the DSL expression problem. Our solution makes use of both virtual inheritance and mixins.

4.1.1 Outline

The rest of this chapter is organized as follows:

- Section 4.2 provides an overview of some of the basic issues that surround multiple inheritance, and some of the main approaches to dealing with these issues that other languages have taken.
- Section 4.3 introduces a mathematical model for mixins: we treat mixins as monotonic functions over modules, and show how the subtype rules for multiple inheritance can be recovered from this definition.
- Section 4.4 discusses the basic obstacles that must be overcome in order to do modular type checking of mixins.
- Section 4.5 introduces the mechanism used by DEEP for detecting name clashes, which are the most difficult obstacle to overcome.
- Section 4.6 gives the syntax, operational semantics, and type theory for the DEEP calculus.
- Section 4.7 describes the current implementation of the DEEP calculus. In particular, it describes how the partial evaluator does code generation.
- Section 4.8 provides some examples of how the calculus can be used to solve

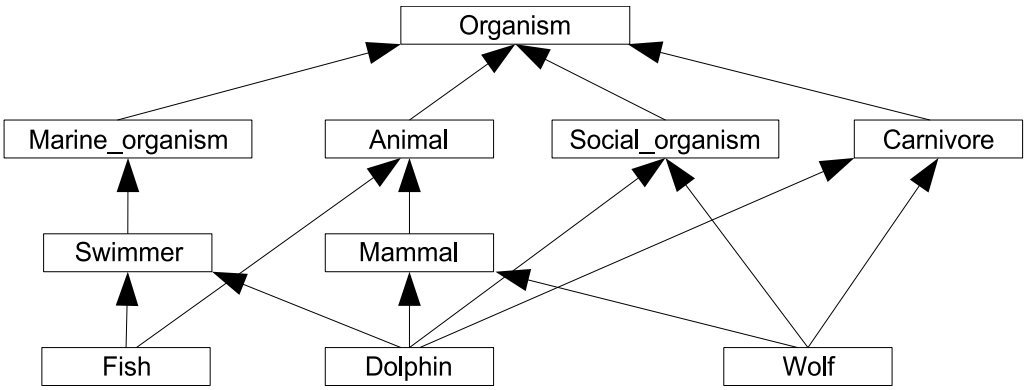


Figure 4.1: A multiple inheritance hierarchy

various problems. We provide two different solutions to the expression problem, and one solution to the DSL expression problem.

- Section 4.9 describes some ideas for future research on DEEP.
- Section 4.10 concludes.

4.2 Multiple inheritance: pitfalls and potential solutions

Inheritance hierarchies are inspired, at least in part, by Linnean taxonomies. A superclass defines behaviors and properties that are common across a set of objects. Each subclass then defines additional properties and behaviors that hold for a more specific set of objects. For example, all mammals are warm-blooded, breath using lungs, bear live young, and nurse their young with milk. In addition to being mammals, dolphins in particular also swim in the ocean, live in social groups, and eat fish.

Multiple inheritance is an intuitive idea because real-world objects can usually be classified into more than one category. For example, in addition to being mammals, dolphins are swimming organisms, social organisms, and carnivores. Their overall appearance and ability to swim is similar to other marine animals, such as sharks and fish, even though sharks and fish are not mammals. Their hunting behavior is similar to other intelligent social carnivores, such as wolves and humans, even though they are not closely related at the genetic level to either of these groups.

A possible inheritance hierarchy based on these concepts is shown in figure 4.1. As can be seen from the figure, real-world taxonomies are often quite complex. There may be a number of properties and behaviors to consider, which are related to each other in complex ways.

Although the idea of inheriting “properties and behavior” from more than one su-

perclass seems intuitive, we have not yet specified what the word *behavior* means, or how, exactly, behavior is inherited. Different object-oriented languages define "behavior" and "inheritance" in different ways. These differences don't matter as much for single inheritance, but become very important when considering multiple inheritance. For a detailed survey of the myriad ways in which inheritance can be, and has been implemented see [Taivalsaari, 1996]. We examine three of the better-known mechanisms here:

1. Interface inheritance
2. Symmetric implementation inheritance
3. Mixin inheritance

The DEEP calculus is based on the third mechanism: mixin inheritance. However, we will discuss the advantages and disadvantages of all three mechanisms in order to explain the tradeoffs that were involved in that design decision.

4.2.1 Interface inheritance

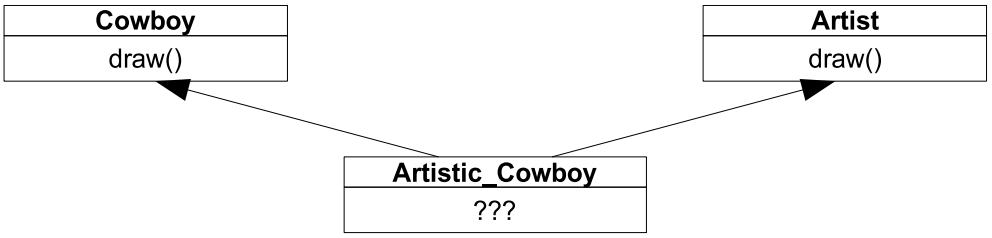
A class defines a set of named methods. The interface of a class is the set of named type signatures for those methods. If we restrict our attention only to interfaces, then the "behavior" of a subclass is easy to define. The interface of a subclass is the union of the interfaces of its superclasses, plus the signatures for any additional methods defined by that particular subclass.

The interface mechanism found in Java captures this particular meaning of the word "behavior". A Java interface only contains method signatures; it may not contain fields or method implementations. Because of this restriction, multiple "inheritance" between interfaces does not present any real technical problems. A Java class may implement any number of interfaces, and interfaces may overlap in arbitrary ways. Subtyping over interfaces follows the standard rules for structural subtyping over records, and multiple inheritance can be modeled with intersection types [Compagnoni and Pierce, 1996].

4.2.1.1 Name clashes

One wrinkle that occassionally crops up with interfaces is the "artistic cowboy problem", which describes a situation in which two unrelated interfaces declare a method with the same name and signature [Rayside and Campbell, 2000]. For example, both

cowboys and artists “draw”, but the meaning of the word “draw” is different; one meaning involves guns, while the other involves pictures.



This particular problem is not too hard to deal with because the clash is due merely to an accident of naming; both versions of draw() happen to have the same name. Consequently, the problem can be resolved by renaming. Mainstream OO languages such as C++ already use *name-mangling*, which automatically renames methods from different classes [Ellis and Stroustrup, 1990]. A similar technique can be used to resolve accidental name clashes of this sort.

4.2.1.2 Advantages of interface inheritance

Interfaces are types, and interface inheritance is merely ordinary subtyping between types. However, a class consists of more than just an interface, because a class includes implementations for methods, whereas an interface does not. Object-oriented *subclassing* thus involves more than ordinary subtyping, because implementations are inherited as well.

It can be argued that subtyping and subclassing are separate mechanisms, and that one should not be confused with the other. In particular, Linnean taxonomies correspond most closely to subtyping, rather than subclassing. The goal of a taxonomy is to organize objects into sets and subsets according to their properties. The standard interpretation of types, in which types denote sets and subtypes denote subsets, thus achieves the goal of a taxonomy quite well. Moreover, the type of an object describes exactly those properties which are relevant to the static type system.

An additional argument in favor of using intersection types, rather than some other mechanism, is the fact that intersection types have three good engineering properties that make them particularly easy to reason about. If & is an intersection type operator, then & is commutative, associative, and idempotent:

$$
\begin{aligned}
t \,\&\, t &= t && \text{(idempotence)} \\
t \,\&\, u &= u \,\&\, t && \text{(commutativity)} \\
s \,\&\, (t \,\&\, u) &= (s \,\&\, t) \,\&\, u && \text{(associativity)}
\end{aligned}
$$

Idempotence ensures that an interface may occur in a composition more than once without any ill effects. Commutativity and associativity ensure that the order in which classes are composed does not matter. As we shall see below, other mechanisms for inheritance do not necessarily have these properties, and it is consequently more difficult to reason about compositions.

4.2.2 Subclassing versus subtyping: a brief digression

Although the interface of an object describes all of the properties and behaviors that are relevant to the static type system, it does not necessarily capture all of the behaviors that are relevant to the programmer. A programmer cares not only about the type signature of a method, but but also about the implementation of that method. After all, the add and subtract methods on integers have the same type signature, but very different meanings.

The standard OO notion of subclassing addresses this particular meaning of the word "behavior" in a way that interfaces alone do not. A subclass inherits the "behavior" of its superclass, because it inherits the implementation of all methods. This analogy is not quite as clear-cut as it could be, because subclasses may arbitrarily override inherited implementations, and there is no way for the language to guarantee that such overriding preserves "behavior" in a meaningful way. Nevertheless, implementation inheritance does provide an indication that a sublass will behave similarly to its superclass at run-time.

Although it is widely used, implementation inheritance remains a somewhat controversial subject. Critics argue that it is easy to abuse, and thus leads to poor code design in practice. Because subclassing is the dominant mechanism for code reuse in object-oriented languages, programmers are often tempted to create a subclass merely to reuse the implementation of an existing class, without paying attention to whether the resulting subclass is a conceptual specialization or not [Szyperski, 1992], [Cook et al., 1990], [America, 1991]. In his survey of inheritance [Taivalsaari, 1996], Taivalsaari writes:

> In fact, the use of inheritance for conceptual specialization seems to be an ideal that is rarely realized.

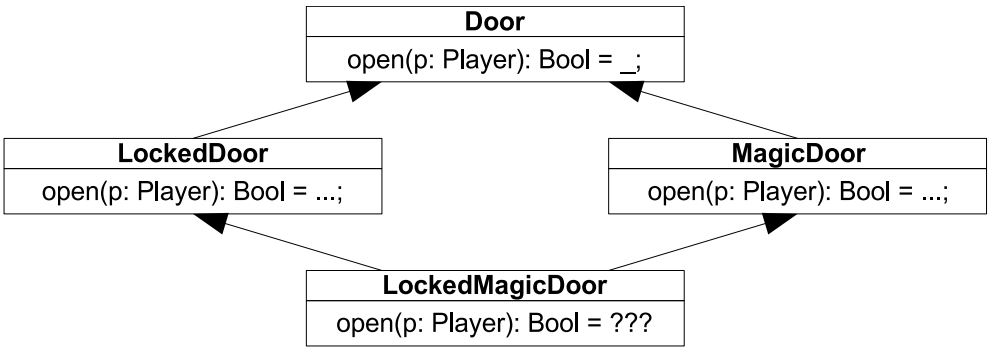


Figure 4.2: The Diamond Problem

The issue at stake here is not whether implementations should be reusable; the need for some mechanism of code reuse is clear. The issue is whether reusing an implementation should necessarily involve creating a subtype. In other words, should subclassing imply subtyping, or should they be regarded as separate mechanisms?

Subtyping places constraints on inheritance that limit the kinds of changes that a subclass is allowed to make. In a proper subtype, it is not possible to remove methods, specialize the argument types of methods, rename methods, or change the visibility of methods from public to private. More flexible module systems, such as those proposed by Bracha [Bracha, 1992], or Ancona and Zucca [Ancona and Zucca, 2002] allow such operations.

Although such flexibility seems desirable at first glance, we argue that these operations are not type-safe on recursive records, and both the DEEP-- and DEEP calculi use a recursive record semantics for modules. In a recursive module, the implementation of any method may refer to *self*. Such an implementation cannot be safely reused in another module, unless *self* in the new module is a subtype of *self* in the old module. The notion of subtyping is thus inextricably intertwined with the notion of reuse; we cannot safely separate subclassing and subtyping into separate mechanisms.

Because of this observation, we will only consider inheritance mechanisms in which subclassing implies subtyping.

4.2.3 The diamond problem

Multiple implementation inheritance is problematic because it gives rise to the infamous “diamond problem” [Bracha, 1992], which has plagued object-oriented languages almost since their inception. The diamond problem occurs when a subclass inherits two different implementations of the same method along different paths.

```
class Door {
 public:
  virtual bool open(Player p) { return true; };
};

class LockedDoor: public Door {
  Key key;

 public:
  virtual bool open(Player& p) {
    return p.hasKey(key);
  }
};

class MagicDoor: public Door {
  Spell openSpell;

 public:
  virtual bool open(Player& p) {
    return p.hasSpell(openSpell);
  }
};

class LockedMagicDoor: public LockedDoor, public MagicDoor {
 public:
  virtual bool open(Player& p) {
    return LockedDoor::open(p) && MagicDoor::open(p);
  }
};
```

Figure 4.3: Symmetric inheritance and manual disambiguation in C++

Figure 4.2 shows an example of the diamond problem, which has been adapted from [Flatt et al., 1998]. Consider a simple game engine, which must deal with different kinds of doors. A locked door can be opened if the player has the appropriate key. A magic door can be opened if the player knows the appropriate spell. Using these two classes, we wish to construct a locked magic door — one which requires both a key and a spell to open.

Note that diamonds are not a problem at the interface level, because the type signature for the open method is the same in both cases. However LockedMagicDoor inherits two different implementations for open, and it must combine those implementations together in some way.

Symmetric implementation inheritance is used by a number of languages. Although

the exact details vary, the basic concept can be found in Ancona and Zucca's module calculus [Ancona and Zucca, 2002], Wells and Vestergaard's m-calculus [Wells and Vestergaard, 2000], *traits* [Schärli et al., 2002], and several mainstream programming languages, including C++ [Ellis and Stroustrup, 1990].

Symmetric implementation inheritance treats both superclasses in the diamond as peers, which means that there is no reason to prefer one implementation over the other. As a result, inheritance diamonds are ambiguous, and the type system will detect and reject such ambiguities as errors. Symmetric multiple inheritance thus requires *manual disambiguation*; the programmer must resolve any conflicts by explicitly combining method implementations by hand.

Figure 4.3 shows C++ code for the doors example. The LockedMagicDoor class resolves the ambiguity by explicitly overriding open, calling open on both of its superclasses, and then combining the results together.

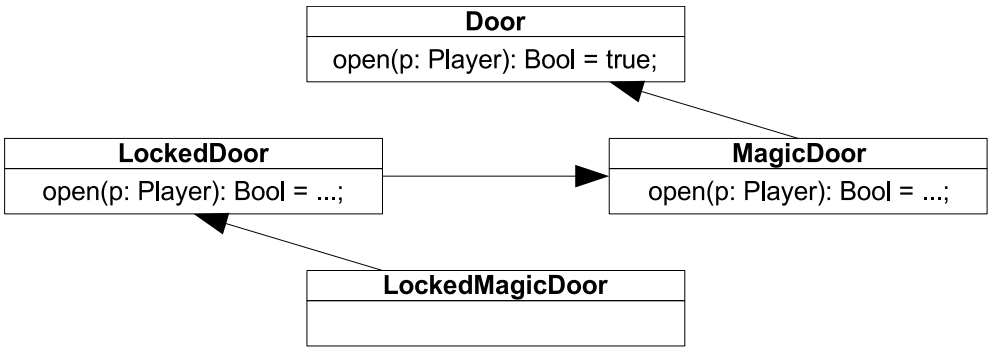
4.2.4 Mixin classes

Mixin classes are an alternative approach to multiple inheritance which became popular in the Common Lisp Object System (CLOS) and its relatives, and have since received a great deal of attention in the literature [Bracha and Cook, 1990] [Flatt et al., 1998] [Ancona et al., 2000]. They can also be implemented in C++ using templates, as shown in Figure 4.4.

Mixin classes make use of a simple but effective techique: they encode classes in such a way that inheritance relationships are not hard-coded [Bracha and Lindstrom, 1992]. LockedDoor and MagicDoor do not inherit from a fixed superclass. Instead, the superclass is a parameter called Super, and LockedDoor and MagicDoor can be instantiated with any arbitrary superclass. Implementations are combined automatically because the implementation of open forwards the call to Super in each case.

The definition of LockedMagicDoor composes LockedDoor and MagicDoor together by rewiring their superclass parameters. In essence, the composition transforms a multiple inheritance hierarchy into a single inheritance hierarchy, as shown in Figure 4.4.

Any multiple inheritance hierarchy can be transformed in this way, so long as the hierarchy is a directed acyclic graph (DAG). Peforming a topological sort of the DAG will yield a linear ordering; a process known as *linearization*. The difficulty with linearization is that this ordering is not unique; there are several different orderings in common use, which have subtly different properties [Barrett et al., 1996].




```
class Door {
 public:
   virtual bool open(Player p) { return true; };
};

template<class Super>
class LockedDoor: public Super {
  Key key;

 public:
   virtual bool open(Player& p) {
     return Super::open(p) && p.hasKey(key);
   }
};

template<class Super>
class MagicDoor: public Super {
  Spell openSpell;

 public:
   virtual bool open(Player& p) {
     return Super::open(p) && p.hasSpell(openSpell);
   }
};

// class LockedMagicDoor extends LockedDoor and MagicDoor
class LockedMagicDoor: public LockedDoor< MagicDoor<Door> > {};
```

Figure 4.4: Mixin-style inheritance in C++

In the C++ example, “super” parameters and linearization are explicit. In languages like CLOS and Dylan, the whole system is implicit. Every class has a hidden “super” parameter, and linearization is performed automatically.

```
template <class T, class K>
class Container {
  virtual T*   find(K key)           { return NULL;}
  virtual void insert(K key, T item) { }
};

template <class T, class K, class Super>
class ConcurrentContainer: public Super {
  Mutex mutex;

  virtual T* find(K key) {
    mutex.lock();
    T* result = Super::find(key);
    mutex.unlock();
    return result;
  }

  virtual void insert(K key, T item) {
    mutex.lock();
    Super::insert(key, item);
    mutex.unlock();
  }
};

template <class T, class K>
class List: public Container<T,K> {
  ...
  virtual T*   find(K key)           { ... }
  virtual void insert(K key, T item) { ... }
};

template <class T, class K>
class ConcurrentList: public ConcurrentContainer<T, K, List<T,K> > { };
```

Figure 4.5: The advantage of mixins

4.2.4.1 Advantages of mixins

The most obvious advantage of mixins is the fact that implementations are combined automatically. Combining implementations by hand is tedious, error-prone, and often requires a detailed knowlege of the exact implementation of all superclasses. With manual disambiguation, the burden of deciding how to combine methods lies on the user of a class library (i.e. the person who is trying to combine LockedDoor and MagicDoor). With mixins, the burden lies on the designer of the class library (i.e. the person who wrote LockedDoor and MagicDoor). This difference makes 3rd-party class libraries

much easier to use.

Even more importantly, there are many cases where disambiguating methods by hand is simply not a viable option. Figure 4.5 shows an example of a situation that is easy to write using mixins, but would be very difficult with manual disambiguation. Container is an abstract class for imperative data structures that support element insertion and lookup. List inherits from Container, and implements find and insert in some way. ConcurrentContainer is a mixin that makes any container safe for use in concurrent programs by using a mutex to control access.

The important thing to notice about this example is the pattern which ConcurrentContainer uses to override each method. ConcurrentContainer does not completely override the implementation, it extends the implementation with locking and unlocking code. Each call to Super is sandwiched between a lock statement and an unlock statement.

Whereas the doors example could be implemented with either mixins or manual disambiguation, this example cannot be, at least not without duplicating code. The lock and unlock statements must be interleaved with the code that does finding or insertion, and manual disambiguation provides no mechanism to interleave implementations in this manner.

4.2.5 Linearization and ordering

Although mixins are powerful, they present their own set of difficulties. The biggest issue is the fact that mixin composition is not symmetric. The behavior of a composition of mixins depends on the order in which they are composed. Going back to the doors example, we see that there are two different ways to compose LockedDoor and MagicDoor:

1. LockedDoor<MagicDoor<Door> >
2. MagicDoor<LockedDoor<Door> >

The first composition looks for a key first, and then looks for a spell, while the second does things the other way around. In the doors example, a change in ordering has no real effect, but the same is not true in general. The ConcurrentContainer example is a case where ordering matters. ConcurrentContainer should always be the outermost mixin in a composition, because it is important to obtain exclusive access to the container before examining or manipulating its state.

Although mixins were initially developed for use with linearization, automatic linearization makes it difficult to control the order in which mixins are composed. In fact, for complex class hierarchies, it can be difficult to even predict the order in which mixins will be composed, not least because there are several different linearization algorithms in use, which have subtly different properties [Barrett et al., 1996].

The DEEP calculus does not perform automatic linearization. Like the C++ example, DEEP requires the order of mixins to be explicitly specified. Linearization is a complex algorithm, and we believe that it should not be part of a core calculus. A real programming language built on top of DEEP could still perform linearization if so desired; the type rules we give are valid regardless of whether composition is explicit or implicit.

4.2.5.1 Duplicates

One advantage of automatic linearization is that it produces compositions in which each mixin class appears at most once. In the absence of linearization, the type system must contend with the possibility of duplicates, e.g. LockedDoor<LockedDoor<Door>>.

Sometimes duplicates can be meaningful, and sometimes they aren't. In the case of doors, LockedDoor<LockedDoor<Door>> might represent a door that has two different locks, and thus requires two different keys to open. Doing the same thing with ConcurrentContainer, however, would just introduce extra overhead by locking and unlocking two mutexes instead of one.

Data fields like mutex are another potential problem. In C++, attempting to add a second copy of ConcurrentContainer would generate an error, because there would be two fields with the same name: mutex. One way around this problem is to share fields with the same name. However, sharing would generate an error at run-time, because the code would then attempt to acquire the same mutex twice, and thus deadlock the program. In order to avoid sharing, mutex must be renamed or otherwise disambiguated [Flatt et al., 1998].

The possibility of duplicates illustrates the fact that although mixins are a powerful tool, they represent a departure from the strict view that inheritance is primarily a way of structuring data according to taxonomic classification. According to the taxonomy at the beginning of the chapter, a dolphin is a mammal. With mixins, however, we could also state that a dolphin is a mammal two or three times over, depending on how many times the mammal mixin appears in a composition.

4.2.6 An algebraic view of mixin composition

The difference between mixins and standard taxonomies can also be illustrated by comparing mixin composition with type intersection. Recall that type intersection (written &) is commutative, associative, and idempotent:

$$\begin{array}{rcll} t \,\&\, t & = & t & \text{(idempotence)} \\ t \,\&\, u & = & u \,\&\, t & \text{(commutativity)} \\ s \,\&\, (t \,\&\, u) & = & (s \,\&\, t) \,\&\, u & \text{(associativity)} \end{array}$$

A mixin is a function from classes to classes. The algebraic operator for mixin composition thus corresponds to function composition (written $\circ$), which is only associative:

$$s \circ (t \circ u) \quad = \quad (s \circ t) \circ u \quad \text{(associativity)}$$

It is not immediately clear from this definition what the subtype rules for mixin composition should be. Since composition is not commutative, subtyping must preserve order. Since composition is not idempotent, subtyping must preserve duplicates. The subtype rules for mixins will thus look quite different from the ones for type intersection.

4.2.7 Summary

Pure interface inheritance, as found in Java, has several advantages. It is simple, easy to reason about, and has a type theory that is relatively well-understood.

Nevertheless, we have rejected the simple interface model for two reasons. First, we wish to support implementation reuse, and interfaces alone are a mechanism for typing, not reuse. There is a practical need for multiple inheritance of implementations. Second, the "behavior" of an object depends on its implementation. Although we cannot formally capture this notion of "behavior" in the type system, we can support it informally by allowing implementations to be inherited, and associating inheritance with subtyping.

There are two competing mechanisms for multiple implementation inheritance: symmetric inheritance, and mixins. We chose mixins, because mixins are more flexible, but they raise serious issues regarding ordering and duplicates. The next section discusses how these issues affect the type system.

4.3 Polarized subtyping

A mixin is a function from classes to classes. However, the subtype rules for mixins and mixin composition are slightly different from ordinary type operators. If F and G are mixins, and A and B are classes or modules, then the following relationships hold:

$$\begin{array}{lcll} F(A) & \leq & A & \text{(refinement)} \\ F(A) & \leq & F(B) \quad \text{if } A \leq B & \text{(monotonicity)} \end{array}$$

The first property, *refinement*, states that a mixin always extends its argument. If a mixin is applied to a class A, then the result will be some subclass of A. The standard definition of higher-order subtyping provides an elegant way to capture this property. We say that an operator F is a mixin if F is a subtype of $(\lambda x \leq t.\, x)$ for some type t. In other words, mixins are subtypes of the identity function on classes.

The second property, *monotonicity* states that if a mixin is passed a more specific class as an argument, then it will produce a more specific class as a result. This property cannot be captured using standard higher-order subtyping. However, if we extend the type system to include *polarized type operators* [Steffen, 1997], then we can capture this property as well.

4.3.1 Polarity of subterms

Polarized subtyping was introduced by Steffen [Steffen, 1997] as a way of distinguishing between type operators at a finer level of detail than is possible in System $F^{\omega}_{\leq}$. Polarized subtyping is based on the idea that each position in a type expression has a particular polarity, which reflects the way in which subtyping is defined over that expression. For example, consider the standard subtype rule for arrow-types, as found in System $F^{\omega}_{\leq}$:

$$\frac{\Gamma \vdash\ T' \leq T \quad U \leq U'}{\Gamma \vdash\ T \to U \leq T' \to U'}$$

Arrow-types are covariant in their result type, but contravariant in their argument type. We say that the result type has a positive polarity, while the argument type has a negative polarity. Subterms that are located within a negative position will have their polarities swapped, as illustrated below. We have labeled the polarities of each subterm, using $^-$ for negative and $^+$ for positive positions:

$$(T_1^+ \to T_2^-)^- \to (U_1^- \to U_2^+)^+$$

In an ordinary type operator, the type variable can occur in any position. It may occur in positive positions, negative positions, or even both positions simultaneously. For example, in the operator $F = \lambda X \leq \mathsf{Top}.\ X \to X$, the variable X occurs in both a positive and a negative position. When such a type operator is applied, its argument is thus *invariant*: we have $F(U) \leq F(U')$ if and only if $U \equiv U'$.

Invariance is a third possible polarity. Following the notation given by Steffen [Steffen, 1997], we shall label invariance $^{\pm}$. (The intuition is that an invariant term occurs in both positive and negative contexts, and thus cannot vary either way.) All subterms that are located in an invariant position must also be invariant, as illustrated by the following term:

$$(T_1^+ \to T_2^-)^- \to F^+(U_1^{\pm} \to U_2^{\pm})^{\pm}$$

4.3.2 Polarized type operators

A *polarized type operator* places a restriction on the positions in which the type variable can appear. If $\lambda X \leq T.\ U$ is a positive operator, then X can only appear in positive positions within U.

Steffen [Steffen, 1997] uses kinds to distinguish the polarities of operators; a positive operator has kind $K_1 \xrightarrow{+} K_2$. The subtype rules make use of kinding information in order to relax the constraint on operator application. Steffen gives the following inference rule for function application, where the judgement $T \leq U : K_1$ means "T is a subtype of U, where both T and U have kind K_1". Notice that the the argument of F is covariant, rather than invariant:

$$\frac{\Gamma \vdash\ F : K_1 \xrightarrow{+} K_2, \quad T \leq U : K_1}{\Gamma \vdash\ F(T) \leq F(U) : K_2}$$

4.3.3 Polarity in DEEP

So far, we have followed (more or less) the notion of polarity used by Steffen in [Steffen, 1997]. Polarity in the DEEP calculus is somewhat simpler, because DEEP does not support contravariance. Contravariance is the cause of several well known problems, as shown in [Pierce, 1994]. Although contravariance is potentially interesting, it complicates the theory, and we do not need it for any of the examples in this thesis. All polarities in DEEP are either positive, or invariant.

The DEEP calculus does not use a kinding system, so we incorporate information on polarities directly into the syntax of terms. By incorporating polarities into the syntax, we avoid the circular dependency between subtyping and kinding that Steffen must contend with. Polarized functions and application are written as follows:

$\lambda^{+}x \leq t.\ u$	positive function
$\lambda^{\pm}x \leq t.\ u$	invariant function
$t(u)^{+}$	positive application
$t(u)^{\pm}$	invariant application

The subtype rules for function application are as follows. For comparison, we show the rules for both positive (covariant) and invariant application side by side. Note that subtyping preserves polarity; positive functions and invariant functions are incomparable, as are positive and invariant applications. The well-formedness rules ensure that positive application can only used to apply a positive function.

$$\frac{\Gamma \vdash\ t \leq t' \quad u \equiv u'}{\Gamma \vdash\ t(u)^{\pm} \leq t'(u')^{\pm}} \qquad \frac{\Gamma \vdash\ t \leq t' \quad u \leq u'}{\Gamma \vdash\ t(u)^{+} \leq t'(u')^{+}}$$

4.3.4 Polarized declarations

In addition to polarized functions, DEEP supports polarized declarations. If d_i is a positive declaration in the module μx extends $t\ \{\overline{d}\}$, then the self-variable x can appear only in positive positions within d_i. The syntax for polarized declarations is as follows. Each label l is tagged with a polarity symbol:

$O\ l^{+} : t$	positive virtual binding
$O\ l^{+} = t$	positive final binding
$t@(u).l^{+}$	positive delegation

As with functions, positive declarations and ordinary declarations are incomparable. A positive declaration can only be overridden with another positive declaration. The subtype rules for delegation are similar to the ones for function application:

$$\frac{\Gamma \vdash\ t \leq t' \quad u \equiv u'}{\Gamma \vdash\ t@(u).l^{\pm} \leq t'@(u').l^{\pm}} \qquad \frac{\Gamma \vdash\ t \leq t' \quad u \leq u'}{\Gamma \vdash\ t@(u).l^{+} \leq t'@(u').l^{+}}$$

Positive declarations are especially useful when dealing with virtual types or nested virtual classes, because the standard OO dot-notation for positive declarations is less restrictive than it is for invariant declarations. Recall that $t.l$ is syntax sugar for $t@(t).l$, so we obtain the following two derived rules:

$$\frac{\Gamma \vdash\ t \equiv t'}{\Gamma \vdash\ t.l^{\pm} \leq t'.l^{\pm}} \qquad \frac{\Gamma \vdash\ t \leq t'}{\Gamma \vdash\ t.l^{+} \leq t'.l^{+}}$$

4.3.5 Subtyping and multiple inheritance

Polarized higher-order subtyping provides enough information to properly model the subtype rules for multiple inheritance. We are interested in studying compositions of the form $F_1(F_2(..\ F_n(A)^+..)^+)^+$, where $F_1..F_n$ are a sequence of mixins, and A is a base class that the mixins are applied to. The following rule for compositions follows directly from the subtype rules given above:

$\Gamma \vdash\ F_1(F_2(..\ F_n(A)^+..)^+)^+ \leq G_1(G_2(..\ G_m(A)^+..)^+)^+$ if and only if:

(1) The sequence $F_1..F_n$ contains all of the mixins in $G_1..G_m$, including duplicates.

(2) The mixins in $F_1..F_n$ are applied in the same order as in $G_1..G_m$.

This rule captures the essential properties of multiple inheritance. A composition inherits behavior from every mixin in the composition, and a composition which contains more mixins is a subtype of one which contains fewer. However, this rule differs from the standard laws for type intersection because it takes duplicates and ordering into account.

The above rule is admissible in deep, because it follows directly from the principles of refinement and monotonicity given before:

$$\begin{array}{ll} F(A)^+ \leq A & \text{if } F \leq \lambda^+ x \leq t.\ x \text{ for some } t \\ F(A)^+ \leq F(B)^+ & \text{if } A \leq B \end{array}$$

The first rule allows us to discard the outermost mixin in a composition, while the second rule, combined with the first, allows us to discard one of the inner mixins. Neither rule allows us to reorder the composition. Taken together, these two rules allow us to write down judgements such as the following, where F, G, H and I are mixins:

$$F(G(H(I(A)^+)^+)^+)^+ \leq F(H(I(A)^+)^+)^+ \leq F(H(A)^+)^+ \leq H(A)^+ \leq A$$

Going back to the doors example at the beginning of this chapter, we find that LockedDoor<MagicDoor<Door>> is in fact a subtype of both LockedDoor<Door> and MagicDoor<Door>, as one might expect.

4.4 Modular type checking for mixins

The most difficult part of the type theory for mixins is performing modular type checking. The term "modular type checking" in this case means verifying that a mixin is well-formed before it is applied.

Mixins were originally developed within the CLOS family of languages (CLOS, Flavors and Loops) [Bobrow et al., 1988], which are not statically typed. They have since been employed within other languages such as Dylan [Barrett et al., 1996] and gbeta [Ernst, 1999b], which are statically typed, but the type systems for these languages have not been fully formalized. The C++ implementations given earlier are not modularly type-checked either; a template is essentially a macro, and is not type-checked at all until it is instantiated with a concrete argument.

A mixin in DEEP looks like the following:

$$\lambda^{+} x \leq t.\ \mu y\ \mathsf{extends}\ x\ \{\overline{d}\}$$

Definitions of this form are not legal in DEEP-- because the module inherits from a variable x, and the DEEP-- type system requires the parent of a module to be statically known. The same restriction can also be found in other OO languages that support modular type-checking, such as Generic Java [Igarashi et al., 1999].

Parents in DEEP-- must be statically known because the type system must guarantee the following three properties. Given a definition μy extends t $\{\overline{d}\}$, the type system must ensure that:

1. Every overriding slot d_l in $\overline{d}$ is a subtype of $t@(y).l$.
2. If d_l is an overriding slot, then l is not final in the superclass t.
3. If d_l is not an overriding slot, then l is not defined in t.

The first requirement is relatively easy to satisfy, and is the subject of this section. The second two requirements are *name clash* requirements. Name clashes are a much harder to satisfy, and are the subject of the next section.

Notice that for all three requirements, we need to know whether a slot d_l is overriding or not. If the parents of a module are statically known, then we can determine whether it is overriding by traversing the chain of parents, and seeing whether any parent previously defined a slot named l; this is what many OO languages do, including Java. However, in the case of mixin classes, the parents of a module are *not* statically known. We therefore require the programmer to specify whether or not a slot is

```
Base = μy extends Top {
  foo(i ≤ Int): Int = i+1;  // method

  MyClass: μz extends Top { // nested class
    bar: Int = 3;
  };
};

Mix(B ≤ Base) = μy extends B {
  override foo(i ≤ Int): Int = i + 2;          // method

  override MyClass: μz extends B@(y).MyClass { // class
    override bar: Int = 4;
  };
};
```

Figure 4.6: Method overriding and extension of nested classes.

overriding, by using the override keyword.

4.4.1 Subtyping of declarations

In order to ensure that inheritance is sound, every overriding slot in a derived module must be a subtype of the corresponding slot in its parent. If the parent is t, and the module self-variable is y, then the "corresponding slot" is $t@(y).l$. This expression denotes the definition of l that the derived module would have inherited from t, if it hadn't overridden l.

In DEEP-- this check will generally fail for mixins because the parent is a variable x. Since x is not statically known, the value of $x@(y).l$ is also unknown, and thus has only trivial subtypes. However, it is not too hard to extend the type system so that this check succeeds for the particular cases that we are interested in supporting.

The two common cases that we wish to support are (1) method overriding, and (2) extending a nested class or module. Both cases are illustrated in Figure 4.6. As shown in the figure, nested classes require no additional support. MyClass extends B@(y).MyClass, so it is clearly a subtype of B@(y).MyClass. The methods foo and bar, however, cannot be verified using the type rules of DEEP--.

Nevertheless, foo and bar can be verified if we add the following two inference rules to the subtype relation. The $\cong$ relation is similar to $\equiv$; we state formally that $t \cong u$ if and only if $t \leq u$ and $u \leq t$.

$$\frac{\Gamma \vdash\ t \leq (\lambda x \leq u.\mathsf{Top})}{\Gamma \vdash\ t \cong \lambda x \leq u.\ t(x)}\ (\text{DS}\eta\text{-FUN}) \qquad \frac{\Gamma \vdash\ t \leq (:u = s)}{\Gamma \vdash\ t \cong (:u = t\$)}(\text{DS}\eta\text{-FIELD})$$

The first rule is simply a statement of classic η-expansion for the λ-calculus. Functions are invariant in their argument type, so if t is a function, then every supertype of t must have the exact same argument type as t. We can use that information to construct a concrete function with the same behavior as t. Fields are similarly invariant in their interface, so if t is a field, then every supertype of t will have the same interface as t. We can use that information to construct a concrete field with the same behavior as t.

Together, these two rules allow us to derive the exact interface of methods in the superclass, even if the superclass is not statically known. For example, we can derive:

```
B@(y).foo ≅ λi ≤ Int. (: Int = B@(y).foo(i)$)
```

This above example also illustrates how congruence is actually used in typechecking. Because the above judgement holds, we can show that the definition of foo in Mix is a subtype of B@(y).foo. The definitions in Mix are thus safe so far as subtyping is concerned; all overriding slots are subtypes of the corresponding slots in the parent.

4.4.1.1 Term congruence

In the λ-calculus, η-rules are typically written as an equivalence relation. We have chosen to write them down as a *congruence* relation instead. Two terms are congruent if each is a subtype of the other, i.e. $t \cong u$ if and only if $t \leq u$ and $u \leq t$. In a type system which has the antisymmetry property, such as System $\lambda_{\lhd}$ or System $F^{\omega}_{\leq}$, congruence implies equivalence. (Antisymmetry states that if $t \leq u$ and $u \leq t$, then $t \equiv u$). DEEP (and DEEP--) are not antisymmetric, because fields allow overriding of both the implementation and the inline clause:

$$\frac{\Gamma \vdash\ t \equiv t'}{\Gamma \vdash\ (:t\ \mathsf{inline}(\overline{s}) = u) \cong (:t'\ \mathsf{inline}(\overline{s'}) = u')}\ (\text{DS-FIELD1})$$

Informally, if t and u are congruent, then t and u are objects which have the exact same interface, but which may have different implementations. The DEEP calculus provides congruence as a formal judgement for determining such "interface equality".

Rule (DSη-FIELD) is a congruence rule, rather than an equivalence rule, for the following reason. If t is a field, then all supertypes of t will have the same interface, so we can easily determine the interface of t. We also know the implementation, which is $t\$$. However, there is no way of determining an appropriate inline clause based on

subtyping alone, because every supertype of t may have a different inline clause. Thus, we leave the inline clause blank, and make the rule a congruence rule, rather than an equivalence rule.

There is also a second, more subtle reason for treating the η-rules as congruence rules rather than equivalence rules, a reason that is also related to inlining. The inline heuristic works by attempting to reduce a set of terms to values. If the terms can be statically reduced to values, then inlining succeeds; if not, then inlining fails. However, η-expansion allows any term with a function or field type to be expanded to a value in one step. The inline mechanism would therefore break if terms were equivalent under η-expansion.

Summary: The equivalence relation is the symmetric and transitive closure of reduction. In other words, equivalence represents *intensional equality*; if two terms are equivalent, then they have a common reduct. The congruence relation permits additional transformations, such as η-expansion and overriding, that preserve the interface of a term, but not its exact implementation.

4.5 Dealing with name clashes

In addition to ensuring that overriding slots are subtypes of ones defined by parents, the type system must ensure that slots defined in a derived module do not cause *name clashes.* As we shall see, name clashes are a major impediment to any attempt to do static modular type-checking of mixins, and there does not seem to be any elegant way of detecting them. We will illustrate the basic problem with a counter-example, as shown in figure 4.7.

This example defines a class A, a class B that inherits from A, and a mixin Mix that extends A. We would expect to be able to apply Mix to B, but doing so will generate two distinct errors. Mix attempts to override a, which is final bound in B, and it adds a new field b, which is already declared in B.

Notice that the problem here is not with the definition of B; B is a perfectly valid subtype of A. Nor is there necessarily a problem with the definition of Mix; Mix seems to extend Super in a perfectly valid manner. The problem only arises when Mix and B are combined together.

In DEEP--, the definition of Mix is not well-formed because the type of Super does not give enough information to guarantee that Mix is applicable to every subtype of A.

```
A = μx extends Top {
  a: Int = 1;
};
B = μx extends A {
  override final a: Int = 2;
  b: Int = 3;
};

Mix(Super ≤ A) = μx extends Super {
  override a: Int = 4;
  b: Bool = true;
};

C = Mix(B);    // composition error
```

Figure 4.7: Two different kinds of name clashes.

In DEEP, the definition of Mix is well-formed. However, in order for Mix to be safe, we need two additional pieces of information:

- Super does not have a final binding for a.
- Super does not have a slot named b.

4.5.1 Inheriting from a virtual class

Name clashes can also arise in another situation: the situation where a derived class inherits from a virtual based class. Since the base class is virtual, it is not statically known. Virtual base classes are thus very similar to mixins in this regard. A name clash can result if the virtual base class is specialized in an incompatible way.

Figure 4.8 uses inheritance to create the same name clashes as in Figure 4.7. M1 is a module with two nested classes, A and B. M2 then inherits from M1, and specializes A. The specialization of A by itself is perfectly sound, but it causes a name clash with the inherited version of B. The error in this example is particularly insidious, because class B is never mentioned in the body of M2. The conflict is "invisible" at the place where it occurs.

4.5.2 Options

There are several ways in which name clashes could potentially be detected, but they all have flaws of various kinds. We will review four possible options, before describing

```
M1 = μx extends Top {
  A: μy extends Top {
    a: Int = 1;
  };
  B: μy extends x.A {
    override a: Int = 5;
    b: Bool = true;
  };
};

M2 = μx extends M1 {
  override A: μy extends M1@(x).A {  // this is valid
    override final a: Int = 2;
    b: Int = 3;
  };
  // inherited definition of B is no longer valid
};
```

Figure 4.8: Inheriting from a virtual class.

---

the solution that we have chosen for DEEP. The four options are:

1. Language restrictions
2. Exact types
3. Generative programming
4. Dynamic linking

4.5.3 Option 1: Language restrictions

One possible approach is to restrict the language in such a way that name clashes cannot occur. The *vc* calculus proposed by Ernst, Ostermann and Cook achieves type-safety in this way, by using the following restrictions [Ernst et al., 2006]:

- All non-overriding methods must have unique names.
- Multiple inheritance is linearized.
- Final bindings are not allowed.
- Parameterized (a.k.a. generic) classes are not allowed.

If all non-overriding method names are globally unique (with uniqueness enforced by name-mangling of some kind), and there are no final bindings, then a name clash can only occur if the same mixin is used in a composition more than once. Linearization

eliminates the possibility of such duplicates, which means that mixins are safe.

However, linearization itself will fail if a composition contains two instances of the same class, but with different parameters, e.g. a collection that inherits from both List(Int) and List(String). It is not safe to eliminate a duplicate unless it has the same type signature as the original. The *vc* calculus avoids this situation by providing no support for parameterized classes. Moreover, since the enclosing module of a class is effectively a hidden parameter, *vc* restricts classes so that they can only inherit from other classes in the same module.

We do not use this approach in DEEP because we regard the lack of parameterized classes, and the lack of final bindings, as being far too restrictive. The practical need for parameterized classes is obvious. As shown in the last chapter, final bindings for type members also play a crucial role in the definition of many modules, and we are not willing to sacrifice that capability.

4.5.4 Option 2: Exact types

As others have pointed out [Fisher and Mitchell, 1995], one of the problems with designing a type-safe language based on delegation is that the type requirements for *using* an object are different from the type requirements for *inheriting* from an object.

The ordinary subtype rules for records and modules make use of *positive type information*. If $B \leq A$, then B has at least all of the methods defined by A. If we define the word "use" to mean "invoke a method", then an object of type B can be "used" in any situation where an object of type A is expected.

However, type-safe inheritance not only requires positive type information, it also requires *negative type information*. Instead of simply listing the methods that a type A is known to have, we must also list the methods that A is known *not* to have. Negative type information has been studied within Haskell or ML-style type systems, using simple records with structural typing and type inference [Gaster and Jones, 1996]. However, DEEP uses recursive records, hierarchically nested records, nominal typing, inheritance, dependent types, and no type inference. It is not at all clear how negative type information could be incorporated into a type system like DEEP.

An alternative is to use types that specify the exact interface of an object. This is the approach used by Fisher and Mitchell in [Fisher and Mitchell, 1995]. They define two different kinds of object. Prototypes have an exact interface type, which means that they can be extended with inheritance, but have only trivial subtypes. Inheritance

does not generate subtypes. Ordinary objects have a rich subtype relation, but cannot be extended with inheritance.

Unfortunately, exact types are not an appropriate way to encode mixins. The whole point of a mixin is that it can be applied to a more specific superclass than the one that was originally declared. Such application is illegal if exact types are strictly enforced.

4.5.5 Option 3: Generative programming

A third approach, and the one which has been used most widely in practice, is code generation. Since mainstream statically typed languages do not support mixins, a number of people have developed various tools which emulate mixins by using source code transformations to generate classes. Examples are approaches based on C++ templates [Smaragdakis and Batory, 2002], feature-oriented programming [Batory and OMalley, 1992] [Batory et al., 2004], and aspect-oriented programming [Kiczales et al., 2001].

None of these generative approaches peform modular type checking. The program generator is a tool which is layered on top of some other language, typically Java. The tool parses the source code of class fragments, merges the fragments syntactically, and then outputs complete class definitions in source-code form. Type-checking does not occur until after all features/aspects/mixins have been composed, when the generated code is handed off to the compiler.

The advantage of code generation is that identifying conflicts is easy, because all necessary information is available at the time when type-checking occurs. The disadvantage of code generation is that separate compilation is impossible, and type errors which occur in generated code can often be quite hard to understand.

4.5.6 Option 4: Dynamic linking

Another way of looking at the problem is to treat name clashes as dynamic errors which are detected at run-time, rather than static type errors that are detected at compile-time. By definition, a static type error is an error which can be detected on the basis of static type information. There are any number of common operations which simply cannot be detected in this way; classic examples are divide by zero, or opening a file that does not exist.

Name clashes cannot be easily detected on the basis of static type information. Thus, it is reasonable to treat them as *dynamic link errors*, rather than static type errors. This interpretation is especially attractive because most modern programming

languages actually have three distinct phases of program execution: compile-time, link-time, and run-time.

- At compile-time, modules are compiled separately to some low-level language such as byte-code or machine code. In a statically typed language, all type-checking happens at this time. Since modules may depended on other modules, there are necessarily inter-module references and dependencies that cannot be resolved at compile-time.
- At link-time, separately compiled modules are loaded into memory and linked together. Inter-module dependencies and references are resolved at this time. Any conflicts between modules, such as name clashes and version mismatches, are detected and resolved at this time.
- At run-time, the low-level language is evaluated by a real or virtual machine.

Detecting name clashes at link-time is relatively simple. Once a set of mixins has been fully composed, we need only traverse the list of parents and ensure that there are no duplicate slots with the same name, and that no final bindings have been overridden. Moreover, this traversal is simple enough to be done on a pre-compiled binary. All we need to know are the names of methods; there is no need for expensive type judgements or access to the original source code.

Although this seems like an elegant way to handle mixins, it suffers from a serious complication: the presence of link errors can cause the type system to make invalid type judgements at compile time. In particular, it is not necessarily safe to inline final bindings with the following rule:

$$\frac{\Gamma \vdash\ t \leq G(\bar{s}) \rightsquigarrow \mu x \text{ extends } t'\ \{\overline{d}\} \qquad O\ l = s \in \overline{d}}{\Gamma \vdash\ t@(u).l \equiv [x \mapsto u]s}$$

In order for this rule to be valid, we must guarantee, at compile-time, that t has only one final binding for l. If there are multiple final bindings for l, then the type system would not be sound, as illustrated below:

```
Mix(i ≤ Int, Super ≤ Top) = μx extends Super {
  foo = i;
};

bar = Mix(3, Mix(4, Top));
```

In this example, the definition of bar is not *well-linked*, and it would thus generate

an error at link-time. At compile time, however, the definition of bar would allow us to derive that $3 \equiv 4$, since bar.foo is equivalent to both numbers! Type safety thus demands that bar be well-linked before it can be used in type judgements.

4.5.7 The DEEP approach

The DEEP calculus uses an approach that is somewhat similar to dynamic linking. Like dynamic linking, our goal is to delay the detection of name clashes. Rather than verifying that there are no name clashes at the point where a module is declared, we verify it later, at the point were the module is used. Unlike dynamic linking, however, we wish to ensure that name clashes are detected at compile-time, during type-checking, in order to avoid the type-safety problems mentioned above.

In order to delay name clash detection, we tag modules with their link status. The syntax $G(\bar{t})_{\surd}$ denotes a module that is statically well-linked. A well-linked module contains no duplicate names, and no final bindings which have been overridden. The syntax $G(\bar{t})_{?}$ denotes a module that is not statically well-linked. For lack of a better term, we shall call such modules "unlinked". Because the module has not been verified as being free of name clashes, it could potentially produce an error.

Because an unlinked module may be erroneous, it is not possible to project a slot from such a module. Before any slots can be projected, the unlinked module must be reduced to a well-linked module. The rule that performs this reduction will traverse the list of parents, and verify that the module contains no name clashes.

4.5.7.1 Well-linked terms

To ensure that linking is static, we extend the notion of "well-linkedness" to cover all terms, not just modules. Well-linked terms come with a guarantee: the evaluation of a well-linked term will not produce link errors. Well-linkedness is thus analogous to well-formedness: the evaluation of a well-formed term is guaranteed not to produce type errors.

Like modules, functions can be tagged with linking information. A well-linked function, written $\lambda^{*}_{\surd} x \leq t.\ u$, must have a well-linked body. Most functions in program will be well-linked. An unlinked function, written $\lambda^{*}_{?} x \leq t.\ u$, may have a body that is unlinked. Unlinked functions allow top-level mixin classes to be wrapped up within ordinary functions, and passed as first-class values, e.g.

```
G              = μx extends Top   { ... };  // base class
```

```
H(super ≤ G) = μx extends super { ... }; // top−level mixin

... λ?g ≤ G. H?(g) ... // wraps H in a λ
```

Function applications are likewise tagged with a linking annotation, depending on whether the function is well-linked or not. A well-linked application is written $t(u)_{\surd}$, while an unlinked application is written $t(u)_{?}$.

With the aid of these annotations, the set of well-linked terms is simple enough to be defined syntactically. A well-linked term, written t^k, is any term which does not have unlinked modules or unlinked applications as subterms, and is defined as follows:

$$
\begin{array}{rl}
t^k, u^k, s^k ::= & \text{(well-linked terms)} \\
& x \\
& \mathsf{Top} \\
& \lambda^*_{\surd} x \leq t^k.\ u^k \\
& \lambda^*_{?} x \leq t^k.\ t \\
& G(\overline{t^k})_{\surd} \\
& : t^k\ \mathsf{inline}(\overline{s^k}) = u^k \\
& t^k (u^k)^*_{\surd} \\
& t^k @ (u^k).l^* \\
& t^k \$
\end{array}
$$

Well-linkedness is preserved under substitution: $[x \mapsto t^k]u^k$ is well-linked for all t^k, u^k. However, showing that well-linkedness is preserved under reduction requires a bit more work. Function application and field extraction are obviously well-linked, as shown below:

- $(\lambda_{\surd} x \leq t^k.\ u^k)(s^k)_{\surd} \longrightarrow [x \mapsto s^k]u^k$
- $(: t^k\ \mathsf{inline}(\overline{s^k}) = u^k)\$ \longrightarrow u^k$

However, delegation does not necessarily produce a well-linked term. According to our current definition of well-linkedness, if a module M is well-linked, then M has no duplicate names, and no final bindings which have been wrongly overridden. However, we have not yet guaranteed that the members of M are also well-linked. The module M may contain virtual classes which inherit from other virtual classes, and those classes must be verified as being well-linked before they can be safely projected from the module.

4.5.7.2 Deep-linking

The well-linkedness judgement for DEEP employs a strategy that we call *deep-linking* (pun intended). In order to verify that a module is well-linked, the type-checker will descend into the module and recursively verify that all nested modules and classes are also well-linked. It will also traverse the list of parents and verify that all inherited definitions are well-linked.

Unlike well-formedness, the well-linkedness judgement does not hold the self-variable of a module to be abstract. Instead, "self" is taken to be equal to the module definition. Whereas the well-formedness check uses a late-binding semantics, the well-linkedness check essentially uses early-binding semantics. The change in binding time is required in order to deal with virtual classes, as illustrated by the following example:

```
MA(m ≤ M) = μx extends Top { a: Int = 0; };
MB(m ≤ M) = μx extends m.A { a: Int = 1; };
NA(n ≤ N) = μx extends MA(n)√ { final a: Int = 2; };

M = μx extends Top {
  A: MA(x)√; // A is a virtual class
  B: MB(x)?; // B inherits from x.A
};

N = μx extends M {
  override A: NA(x)√;  // invalidates B
};

... M()√ ...  // okay, M is well-linked
... N()√ ...  // error, N is not well-linked
```

For clarity, this example uses the full formal syntax of DEEP, in which all modules have been lifted to top-level definitions. M is a module with two virtual classes, A and B, where B inherits from A. Because B inherits from a virtual class, it must be declared as unlinked.

The well-formedness check for M is done at the point where M is declared, but the well-linkedness check is not. The well-linkedness check is not done until the point where M is used, which comes later in the code.

The verification that M is well-linked succeeds, because B is well-linked under the assumption that x = M; we say that B is well-linked within M. However, the fact that B is well-linked within M does not mean that it is well-linked within N. The verification of N will fail; B is not well-linked under the assumption that x = N, because it attempts

to override a final binding.

This change in binding times illustrates the reason why well-formedness and well-linkedness must be considered as separate judgements. Every term that is well-formed within the definition of M is also well-formed within N; this property is ensured because the body of M is type-checked under the assumption that $\mathsf{x} \leq \mathsf{M}$.

In contrast, terms that are well-linked within M are not necessarily well-linked within N. Linking is done on a module-by-module basis. There is no way to know whether a particular module composition will be well-linked, short of actually performing the composition and verifying the result.

This particular strategy for finding name clashes bears some resemblance to generative programming, because naming errors are not discovered until well after a module has been declared. It is an improvement, however, because all other type errors are detected immediately.

4.5.7.3 Static detection of name clashes

Although they are separate judgements, well-formedness and well-linkedness are intertwined. A module $G(\bar{t})_{\surd}$ is only well-formed if it is well-linked (in contrast with $G(\bar{t})_{?}$, which can be well-formed without being well-linked). In addition, a program is only well-formed if the main clause of the program is well-linked.

For terms, the judgement $t \equiv u^k$ demonstrates well-linkedness. A well linked term must either have the syntactic form u^k that was described earlier, or the partial evaluator must be able to reduce it to some u^k. Reduction is necessary to eliminate unlinked function applications, or to reduce an unlinked module to a well-linked module by performing deep-linking.

Partial evalution allows us to interleave type checking and linking. Since partial evaluation is part of subtyping, and linking is part of partial evaluation, linking can be done incrementally, on an "as needed" basis. By interleaving type checking and linking, we are able to avoid the type safety issues mentioned in Section 4.5.6.

Nevertheless, in the case of whole programs, our strategy corresponds closely to the separate linking phase used by many languages, in which linking happens after compile-time, but before run-time. A program in DEEP has the syntax $\mathsf{let}\ G_1 = M_1, .. G_n = M_n\ \mathsf{in}\ t$, where $G_i = M_i$ are a sequence of top-level module definitions. The modules must be well-formed, but they are not necessarily well-linked. (In the case of mixin modules, they most definitely will not be well-linked, because mixins must be composed before they can be verified as well-linked.)

In practical programming, the modules $G_1..G_n$ would be defined in separate files and compiled separately. The main program clause t would then be used to link selected modules together and form a running program, as shown in the following example:

```
let G1 = μx {
      foo: Int = _;
      bar: Int = _;
      main: Int = x.foo + x.bar;
    }
    G2(g ≤ G1) = μx extends g {
      override foo: Int = 1;
    }
    G3(g ≤ G1) = μx extends g {
      override bar: Int = 1;
    }
in G3(G2(G1)))√.main
```

The composition $\mathsf{G3(G2(G1)))}_{\surd}$ must be verified as well-linked before main is evaluated. The linking step will descend into the composition and ensure that every nested member is also well-linked, although in this case foo, bar, and main are trivial. Well-linkedness is preserved under reduction, so once the linking phase is complete, further evaluation cannot produce any link errors.

4.5.7.4 Static mixin composition

One consequence of our design decision is that mixin composition in DEEP is static, rather than dynamic. It is not possible to create new mixin compositions on the fly; all compositions must be created and verified at compile-time. The DEEP calculus differs in this regard from some other proposed mechanisms, such as [Buchi and Weck, 2000] and [Ostermann, 2002].

The reason that mixin composition is static is because it relies on partial evaluation. The partial evaluator will take a program that is unlinked, and partially evaluate it to one that is well-linked. All mixin compositions must be performed by the partial evaluator before run-time. If there are any remaining compositions that have not yet been performed, then partial evaluation will fail to yield a program that is completely well-linked, and the type checker (or linker) will signal an error.

x, y, z	variables	$d, e, f ::=$	declarations
ℓ, l, m	slot names	$\quad O\ l^* \doteq t$	labeled term
G, H	global variables	$O ::= \mathsf{def} \mid \mathsf{override}$	modifier
		$* ::= \pm \mid +$	polarity
Program ::=	program	$\doteq ::= : \mid =$	virtual/final
$\quad \mathsf{let}\ \overline{G} = \overline{M}\ \mathsf{in}\ t$	letrec	$\sigma ::= \surd \mid ?$	linked/unlinked
$M, N ::=$	module definition	$v, w ::=$	values
$\quad \mu x\ \mathsf{extends}\ t\ \{\overline{d}\}$	module	$\quad \mathsf{Top}$	Top-type
$\quad \lambda^* x \leq t.\ M$	param. module	$\quad \lambda^*_\sigma x \leq t.\ u$	function
		$\quad G(\overline{t})_\surd$	linked module
$s, t, u ::=$	terms	$\quad : t\ \mathsf{inline}(\overline{s}) = u$	field
$\quad x$	variable		
$\quad \mathsf{Top}$	Top-type	$\Gamma ::=$	contexts
$\quad \lambda^*_\sigma x \leq t.\ u$	function	$\quad \emptyset$	empty context
$\quad G(\overline{t})_\sigma$	module	$\quad \Gamma, x \leq t$	upper bound
$\quad : t\ \mathsf{inline}(\overline{s}) = u$	field		
$\quad t(u)^*_\sigma$	apply	$\lhd ::=$	type relation
$\quad t@(u).l^*$	delegate	$\quad \leq$	subtype
$\quad t\$$	extract	$\quad \cong$	congruence
		$\quad \equiv$	equivalence

Notation:

- $\overline{t}$ and $\overline{d}$ denote a sequence of zero or more terms or declarations, respectively.
- $\mathrm{dom}(\overline{d})$ denotes the set of labels in the sequence of declarations $\overline{d}$.
- $GT(G_i)$ denotes the module named G_i in the program: let $\overline{G} = \overline{M}$ in t.

Evaluation Context:

$$C ::= [] \mid C(t)^*_\sigma \mid t(C)^*_\sigma \mid C@(t).l^* \mid t@(C).l^* \mid C\$ \mid G(\overline{t}, C, \overline{u})_\sigma$$
$$\mid \lambda^*_\sigma x \leq C.\ t \mid \lambda^*_\sigma x \leq t.\ C \mid : C\ \mathsf{inline}(\overline{s}) = u \mid : t\ \mathsf{inline}(\overline{s}, C, \overline{s'}) = u \mid : t\ \mathsf{inline}(\overline{s}) = C$$

Reduction: $\boxed{t \longrightarrow t'}$

$$(\lambda^*_\sigma x \leq t.\ u)(s)^*_\sigma \longrightarrow [x \mapsto s]u \quad \text{(E-APP)}$$

$$\frac{G(\overline{t}) \rightsquigarrow \mu x\ \mathsf{extends}\ t'\ \{\overline{d}\} \quad O\ l^* \doteq u\ \in \overline{d}}{G(\overline{t})_\surd@(s).l^* \longrightarrow [x \mapsto s]u} \quad \text{(E-DLG1)}$$

$$\frac{G(\overline{t}) \rightsquigarrow \mu x\ \mathsf{extends}\ t'\ \{\overline{d}\} \quad l \notin \mathrm{dom}(\overline{d})}{G(\overline{t})_\surd@(s).l^* \longrightarrow t'@(s).l^*} \quad \text{(E-DLG2)}$$

$$(: t\ \mathsf{inline}(\overline{s}) = u)\$ \longrightarrow u \quad \text{(E-EXT)}$$

$$G(\overline{s})_? \longrightarrow G(\overline{s})_\surd \quad \text{(E-LINK)}$$

$$\frac{t \longrightarrow t'}{C[t] \longrightarrow C[t']} \quad \text{(E-CONG)}$$

Global module lookup: $\boxed{G(\overline{t}) \rightsquigarrow M}$

$$G() \rightsquigarrow GT(G) \quad \text{(E-LOOK1)}$$

$$\frac{G(\overline{t}) \rightsquigarrow \lambda^* x \leq s.\ M}{G(\overline{t}, u) \rightsquigarrow [x \mapsto u]M} \quad \text{(E-LOOK2)}$$

Figure 4.9: The DEEP calculus — syntax and operational semantics

Well-subtyping: $\boxed{\Gamma \vdash\ t \lhd_{\text{wf}} u}$

$$\frac{\Gamma \vdash\ t\ \text{wf}, \quad u\ \text{wf}, \quad t \lhd u}{\Gamma \vdash\ t \lhd_{\text{wf}} u} \quad \text{(W-SUB)}$$

Subtyping & equivalence: $\boxed{\Gamma \vdash\ t \lhd u}$

$$\frac{\Gamma \vdash\ s \lhd t, \quad t \lhd u, \quad t\ \text{wf}}{\Gamma \vdash\ s \lhd u} \quad \text{(DS-TRANS)}$$

$$\frac{\Gamma \vdash\ u \equiv t}{\Gamma \vdash\ t \equiv u} \qquad \frac{\Gamma \vdash\ u \cong t}{\Gamma \vdash\ t \cong u} \quad \text{(DS-SYM1-2)}$$

$$\frac{\Gamma \vdash\ t \equiv u}{\Gamma \vdash\ t \cong u} \quad \text{(DS-CONG)} \qquad \frac{\Gamma \vdash\ t \cong u}{\Gamma \vdash\ t \leq u} \quad \text{(DS-EQ)}$$

$$\frac{\begin{array}{c}\Gamma \vdash\ t \equiv t' \\ \Gamma, x \leq t \vdash\ u \lhd u'\end{array}}{\Gamma \vdash\ \lambda^*_\sigma x \leq t.\ u\ \lhd\ \lambda^*_\sigma x \leq t'.\ u'} \quad \text{(DS-FUN)}$$

$$\frac{G() \rightsquigarrow M}{\Gamma \vdash\ G()_\sigma \equiv G()_\sigma} \quad \text{(DS-MOD1)}$$

$$\frac{\Gamma \vdash\ G(\overline{t})_\sigma \lhd G(\overline{t'})_\sigma \quad u \equiv u'}{\Gamma \vdash\ G(\overline{t}, u)_\sigma \lhd G(\overline{t'}, u')_\sigma} \quad \text{(DS-MOD2)}$$

$$\frac{\begin{array}{c}G(\overline{t}) \rightsquigarrow \lambda^+ x \leq s.\ M \\ \Gamma \vdash\ G(\overline{t})_\sigma \leq G(\overline{t'})_\sigma, \quad u \leq u'\end{array}}{\Gamma \vdash\ G(\overline{t}, u)_\sigma \leq G(\overline{t'}, u')_\sigma} \quad \text{(DS-MOD2+)}$$

$$\Gamma \vdash\ \text{Top} \equiv \text{Top} \quad \text{(DS-TOP)}$$

$$\frac{x \in \text{dom}(\Gamma)}{\Gamma \vdash\ x \equiv x} \quad \text{(DS-VAR)}$$

$$\frac{\Gamma \vdash\ t \lhd t', \quad u \equiv u'}{\Gamma \vdash\ t(u)^*_\sigma \lhd t'(u')^*_\sigma} \quad \text{(DS-APP)}$$

$$\frac{\Gamma \vdash\ t \leq t', \quad u \leq u'}{\Gamma \vdash\ t(u)^+_\sigma \leq t'(u')^+_\sigma} \quad \text{(DS-APP+)}$$

$$\frac{\Gamma \vdash\ t \lhd t', \quad u \equiv u'}{\Gamma \vdash\ t@(u).l^* \lhd t'@(u').l^*} \quad \text{(DS-DLG)}$$

$$\frac{\Gamma \vdash\ t \leq t', \quad u \leq u'}{\Gamma \vdash\ t@(u).l^+ \leq t'@(u').l^+} \quad \text{(DS-DLG+)}$$

$$\frac{\Gamma \vdash\ t \equiv u}{\Gamma \vdash\ t\$ \equiv u\$} \quad \text{(DS-EXT)}$$

$$\Gamma \vdash\ t \leq \text{Top} \quad \text{(DS-ETOP)}$$

$$\frac{x \leq t \in \Gamma}{\Gamma \vdash\ x \leq t} \quad \text{(DS-EVAR)}$$

$$\frac{G(\overline{t}) \rightsquigarrow \mu x\ \text{extends}\ u\ \{\overline{d}\}}{\Gamma \vdash\ G(\overline{t})_\sigma \leq u} \quad \text{(DS-EINH)}$$

$$\frac{\Gamma \vdash\ t \leq_{\text{wf}} (:u\ \text{inline}(\overline{s}) = u')}{\Gamma \vdash\ t\$ \leq u} \quad \text{(DS-EEXT1)}$$

$$\frac{\Gamma \vdash\ t \equiv t'}{\Gamma \vdash\ (:t\ \text{inline}(\overline{s}) = u) \cong (:t'\ \text{inline}(\overline{s'}) = u')} \quad \text{(DS-FIELD1)}$$

$$\frac{\Gamma \vdash\ t \equiv t', \quad u \equiv u', \quad \overline{s} \equiv \overline{s'}}{\Gamma \vdash\ (:t\ \text{inline}(\overline{s}) = u) \equiv (:t'\ \text{inline}(\overline{s'}) = u')} \quad \text{(DS-FIELD2)}$$

Figure 4.10: Declarative subtyping for DEEP

4.6 The DEEP calculus: syntax and semantics

The syntax and operational semantics of DEEP are shown in Figure 4.9. The declarative subtyping rules are shown in figures 4.10 and 4.11. The well-formedness rules are shown in figure 4.12, and the well-linkedness rules are shown in 4.13.

The operational semantics of DEEP is identical to the semantics for DEEP--. There

Equivalence (reduction): $\boxed{\Gamma \vdash\ t \equiv t'}$

$$\Gamma \vdash (\lambda^*_\sigma x \leq t.\ u)(s^k)^*_\sigma \equiv [x \mapsto s^k]u \quad \text{(DS-EAPP)}$$

$$\frac{G(\overline{t^k}) \rightsquigarrow \mu x \text{ extends } t' \ \{\overline{d}\} \quad O\ l^* \doteqdot u \in \overline{d}}{\Gamma \vdash\ G(\overline{t^k})_{\sqrt{}}@(s^k).l^* \equiv [x \mapsto s^k]u} \quad \text{(DS-EDLG1)}$$

$$\frac{G(\overline{t^k}) \rightsquigarrow \mu x \text{ extends } t' \ \{\overline{d}\} \quad l \notin \mathrm{dom}(\overline{d})}{\Gamma \vdash\ G(\overline{t^k})_{\sqrt{}}@(s^k).l^* \equiv t'@(s^k).l^*} \quad \text{(DS-EDLG2)}$$

$$\frac{\overline{v} \text{ not empty}}{\Gamma \vdash (:t\ \mathsf{inline}(\overline{v}) = u)\$ \equiv u} \quad \text{(DS-EEXT)}$$

$$\frac{\Gamma \vdash\ t^k \leq_{\mathrm{wf}} G(\overline{t'})_\sigma \quad G(\overline{t'}) \rightsquigarrow \mu x \text{ extends } s' \ \{\overline{d}\} \quad O\ l^* = u \in \overline{d}}{\Gamma \vdash\ t^k@(s^k).l^* \equiv [x \mapsto s^k]u} \quad \text{(DS-EFINAL)}$$

$$\frac{\Gamma \vdash\ G(\bar{t})\ \mathrm{wlk}}{\Gamma \vdash\ G(\bar{t})_? \equiv G(\bar{t})_{\sqrt{}}} \quad \text{(DS-ELINK)}$$

$$\frac{G(\overline{t^k}) \rightsquigarrow \mu x \text{ extends } t' \ \{\overline{d}\} \quad O\ l^* \doteqdot u \in \overline{d}}{\Gamma \vdash\ G(\overline{t^k})_?@(s^k).l^* \leq [x \mapsto s^k]u} \quad \text{(DS-EDLGSUB)}$$

Congruence (η): $\boxed{\Gamma \vdash\ t \cong t'}$

$$\frac{\Gamma \vdash\ t \leq_{\mathrm{wf}} \lambda^*_\sigma x \leq s.\ \mathsf{Top}}{\Gamma \vdash\ t \cong \lambda^*_\sigma x \leq s.\ t(x)^*_\sigma} \quad \text{(DS-ηFUN)}$$

$$\frac{\Gamma \vdash\ t \leq_{\mathrm{wf}} (:u\ \mathsf{inline}(\overline{s}) = u')}{\Gamma \vdash\ t \cong (:u\ \mathsf{inline}() = t\$)} \quad \text{(DS-ηFIELD)}$$

Figure 4.11: Type equivalence and congruence in DEEP

---

are no new concepts or reduction rules. The difference between DEEP and DEEP-- is strictly confined to the type system. The syntax of DEEP extends DEEP-- with a number of additional type annotations, as described in the previous sections. There are three major changes:

1. Functions and declarations are now tagged with a polarity symbol $*$, as are applications and delegation. The $*$ symbol is a meta-variable which ranges over $+$

Context well-formedness: $\boxed{\Gamma \text{ wf}}$

$$\frac{}{\emptyset \text{ wf}} \quad \text{(W-GAM1)}$$

$$\frac{\Gamma \text{ wf}, \quad x \notin \text{dom}(\Gamma) \quad \Gamma \vdash t \text{ wf}}{\Gamma, x \leq t \text{ wf}} \quad \text{(W-GAM2)}$$

Term well-formedness: $\boxed{t \text{ wf}}$

$$\frac{\Gamma \text{ wf} \quad x \in \text{dom}(\Gamma)}{\Gamma \vdash x \text{ wf}} \quad \text{(W-VAR)}$$

$$\frac{\Gamma \text{ wf}}{\Gamma \vdash \mathsf{Top} \text{ wf}} \quad \text{(W-TOP)}$$

$$\frac{\Gamma, x \leq t \vdash u \text{ wf} \quad x \notin \text{npvars}(*, u)}{\Gamma \vdash \lambda^*_\sigma x \leq t.\, u \text{ wf}} \quad \text{(W-FUN)}$$

$$\frac{G() \rightsquigarrow M}{\Gamma \vdash G()_? \text{ wf}} \quad \text{(W-MOD1)}$$

$$\frac{\Gamma \vdash G(\bar{t})_? \text{ wf} \quad G(\bar{t}) \rightsquigarrow (\lambda^* x \leq s.\, M) \quad \Gamma \vdash u \leq_{\text{wf}} s}{\Gamma \vdash G(\bar{t}, u)_? \text{ wf}} \quad \text{(W-MOD2)}$$

$$\frac{\Gamma \vdash \bar{s} \text{ wf}, \quad u \leq_{\text{wf}} t}{\Gamma \vdash (: t \ \mathsf{inline}(\bar{s}) = u) \text{ wf}} \quad \text{(W-FIELD)}$$

$$\frac{\Gamma \vdash t \leq_{\text{wf}} (\lambda^*_\sigma x \leq s.\, \mathsf{Top}) \quad \Gamma \vdash u \leq_{\text{wf}} s}{\Gamma \vdash t(u)^*_\sigma \text{ wf}} \quad \text{(W-APP)}$$

$$\frac{\Gamma \vdash t \leq_{\text{wf}} G(\bar{s})_\sigma \quad G(\bar{s}) \rightsquigarrow \mu x \ \mathsf{extends}\ t' \ \{\bar{d}\} \quad O\ l^* \doteqdot u' \in \bar{d} \quad \Gamma \vdash u \leq_{\text{wf}} t}{\Gamma \vdash t@(u).l^* \text{ wf}} \quad \text{(W-DLG)}$$

$$\frac{\Gamma \vdash t \leq_{\text{wf}} (: u \ \mathsf{inline}(\bar{s}) = u')}{\Gamma \vdash t\$ \text{ wf}} \quad \text{(W-EXT)}$$

$$\frac{\Gamma \vdash G(\bar{t})_? \text{ wf} \quad \Gamma \vdash G(\bar{t}) \text{ wlk}}{\Gamma \vdash G(\bar{t})_\surd \text{ wf}} \quad \text{(W-MOD}\surd\text{)}$$

Program well-formedness:

$$\frac{\overline{G} :: \emptyset \vdash \overline{M} \text{ wf} \quad \emptyset \vdash t \equiv_{\text{wf}} s^k \quad \text{There are no illegal cycles.}}{\mathsf{let}\ \overline{G} = \overline{M}\ \mathsf{in}\ t \text{ wf}} \quad \text{(W-PROG)}$$

Module wf: $\boxed{G(\bar{s}) :: \Gamma \vdash M \text{ wf}}$

$$\frac{G(\bar{s}, x) :: \Gamma, x \leq t \vdash M \text{ wf} \quad x \notin \text{npvars}(*, M)}{G(\bar{s}) :: \Gamma \vdash \lambda^* x \leq t.\, M \text{ wf}} \quad \text{(W-MFUN)}$$

$$\frac{\Gamma \vdash t \text{ wf} \quad \Gamma, x \leq G(\bar{s})_? \vdash \bar{d} \text{ wf} \quad \bar{d} \text{ has no duplicate labels.}}{G(\bar{s}) :: \Gamma \vdash \mu x \ \mathsf{extends}\ t \ \{\bar{d}\} \text{ wf}} \quad \text{(W-MDEF)}$$

Declaration well-formedness: $\boxed{\Gamma \vdash d \text{ wf}}$

$$\frac{x \notin \text{npvars}(*, \mathsf{def}\ l^* \doteqdot t) \quad \Gamma, x \leq G(\bar{s})_\sigma \vdash t \text{ wf}}{\Gamma, x \leq G(\bar{s})_\sigma \vdash \mathsf{def}\ l^* \doteqdot t \text{ wf}} \quad \text{(W-DECL)}$$

$$\frac{x \notin \text{pvars}(*, \mathsf{override}\ l^* \doteqdot t) \quad G(\bar{s}) \rightsquigarrow \mu y \ \mathsf{extends}\ u \ \{\bar{d}\} \quad \Gamma, x \leq G(\bar{s})_\sigma \vdash t \leq_{\text{wf}} u@(x).l^*}{\Gamma, y \leq G(\bar{s})_\sigma \vdash \mathsf{override}\ l^* \doteqdot t \text{ wf}} \quad \text{(W-ODECL)}$$

Figure 4.12: Well-formedness rules for DEEP

Well-linked terms:

$$
\begin{array}{rll}
t^k, u^k, s^k ::= & \text{(well-linked terms)} \\
& x \\
& \mathsf{Top} \\
& \lambda^*_{\surd} x \le t^k.\ u^k \\
& \lambda^*_{?} x \le t^k.\ t \\
& G(\overline{t^k})_{\surd} \\
& :t^k\ \mathsf{inline}(\overline{s^k}) = u^k \\
& t^k (u^k)^*_{\surd} \\
& t^k @ (u^k).l^* \\
& t^k \$
\end{array}
$$

Well-linked modules: $\boxed{\Gamma \vdash\ G(\overline{t})\ \text{wlk}}$

$$
\frac{\begin{array}{c} G(\overline{t}) \rightsquigarrow \mu x\ \mathsf{extends}\ u\ \{\overline{d}\} \\ (\mathsf{def}\ l \doteqdot s) \in \overline{d} \quad \text{implies } \Gamma \vdash\ u.l\ \text{undefined} \\ (\mathsf{override}\ l \doteqdot u) \in \overline{d}\ \text{implies } \Gamma \vdash\ u.l\ \text{virtual} \\ \Gamma \vdash\ G(\overline{t})_{\surd} @ (G(\overline{t}))\ \text{valid} \end{array}}{\Gamma \vdash\ G(\overline{t})\ \text{wlk}}
$$

$$
\Gamma \vdash\ \mathsf{Top} @ (G(\overline{s}))\ \text{valid}
$$

$$
\frac{\begin{array}{c} \Gamma \vdash\ t \equiv H(\overline{u})_{\surd} \\ H(\overline{u}) \rightsquigarrow \mu x\ \mathsf{extends}\ t'\ \{\overline{O\ l \doteqdot u}\} \\ \forall u_i.\ \Gamma \vdash\ [x \mapsto G(\overline{s})_{\surd}] u_i \equiv s_i^k \\ \Gamma \vdash\ t' @ (G(\overline{s}))\ \text{valid} \end{array}}{\Gamma \vdash\ t @ (G(\overline{s}))\ \text{valid}}
$$

$$
\Gamma \vdash\ \mathsf{Top}.l\ \text{virtual}
$$

$$
\Gamma \vdash\ \mathsf{Top}.l\ \text{undefined}
$$

$$
\frac{\begin{array}{c} \Gamma \vdash\ t \equiv G(\overline{s})_{\surd} \\ G(\overline{s}) \rightsquigarrow \mu x\ \mathsf{extends}\ t'\ \{\overline{d}\} \\ O\ l^* = u\ \notin \overline{d} \quad \Gamma \vdash\ t'.l\ \text{virtual} \end{array}}{\Gamma \vdash\ t.l\ \text{virtual}}
$$

$$
\frac{\begin{array}{c} \Gamma \vdash\ t \equiv G(\overline{s})_{\surd} \\ G(\overline{s}) \rightsquigarrow \mu x\ \mathsf{extends}\ t'\ \{\overline{d}\} \\ l \notin \mathrm{dom}(\overline{d}) \quad \Gamma \vdash\ t'.l\ \text{undefined} \end{array}}{\Gamma \vdash\ t.l\ \text{undefined}}
$$

$$
\begin{array}{rclrcl}
\mathrm{npvars}(\pm, t) & = & \emptyset & \mathrm{npvars}(\pm, M) & = & \emptyset \\
\mathrm{npvars}(+, t) & = & \mathrm{IV}(t) & \mathrm{npvars}(+, M) & = & \mathrm{IV}(M)
\end{array}
$$

$$
\begin{array}{lcll}
\mathrm{IV}(x) & = & \emptyset & \\
\mathrm{IV}(\mathsf{Top}) & = & \emptyset & \\
\mathrm{IV}(t(u)^{\pm}) & = & \mathrm{IV}(t) \cup \mathrm{fv}(u) & \\
\mathrm{IV}(t(u)^{+}) & = & \mathrm{IV}(t) \cup \mathrm{IV}(u) & \\
\mathrm{IV}(t @ (u).l^{\pm}) & = & \mathrm{IV}(t) \cup \mathrm{fv}(u) & \\
\mathrm{IV}(t @ (u).l^{+}) & = & \mathrm{IV}(t) \cup \mathrm{IV}(u) & \\
\mathrm{IV}(\lambda^*_{\sigma} x \le t.\ u) & = & \mathrm{fv}(t) \cup \mathrm{IV}(u) - \{x\} & \\
\mathrm{IV}(:t\ \mathsf{inline}(\overline{s}) = u) & = & \mathrm{fv}(t) & \\
\mathrm{IV}(G()) & = & \emptyset & \\
\mathrm{IV}(G(\overline{t}, u)) & = & \mathrm{IV}(G(\overline{t})) \cup \mathrm{fv}(u) & \text{if } G(\overline{t}) \rightsquigarrow \lambda x^{\pm} \le s.\ M \\
\mathrm{IV}(G(\overline{t}, u)) & = & \mathrm{IV}(G(\overline{t})) \cup \mathrm{IV}(u) & \text{if } G(\overline{t}) \rightsquigarrow \lambda x^{+} \le s.\ M
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{IV}(\lambda^* x \le t.\ M) & = & \mathrm{fv}(t) \cup \mathrm{IV}(M) - \{x\} \\
\mathrm{IV}(\mu x\ \mathsf{extends}\ t\ \{\overline{d}\}) & = & \mathrm{IV}(t) \cup \mathrm{IV}(\overline{d}) - \{x\} \\
\mathrm{IV}(O\ l^* : t) & = & \mathrm{IV}(t) \\
\mathrm{IV}(O\ l^* = t) & = & \mathrm{fv}(t)
\end{array}
$$

The set $\mathrm{IV}(t)$ is the set of free variables that are in invariant positions in t. (We write $\mathrm{IV}(t)$ in capital letters to distinguish it visually from $\mathrm{fv}(t)$, which is the set of all free variables in t.)

Figure 4.13: Well-linkedness rules and invariant variables in DEEP

and $\pm$.

2. Functions, applications, and modules are now tagged with as being either unlinked ? or well-linked $\surd$. The σ meta-variable ranges over ? and $\surd$.
3. There are now three type relations: $\leq$, $\cong$, and $\equiv$. The $\lhd$ symbol ranges over all three.

The subtyping and well-formedness rules for DEEP-- have been extended accordingly. On the subtyping side, (DS-MOD2+), (DS-APP+), and (DS-DLG+) are new rules that take polarity information into account. On the well-formedness side, (W-FUN), (W-MFUN), (W-DECL), and (W-ODECL) use the "npvars" function to ensure that positive functions and declarations do not have free variables in invariant positions.

The basic properties of congruence are defined by rules (DS-SYM2) and (DS-CONG). (DS-FIELD1) has been changed from a subtype rule to a congruence rule, and (DS-ηFUN) and (DS-ηFIELD) are new rules which perform η-expansion.

Linking is done by rule (DS-ELINK), which converts an unlinked module to a well-linked module by using the well-linkedness judgement: $G(\bar{t})$ wlk. All of the equivalence rules are now confined to well-linked terms; a term must be well-linked before it can be used. The one exception is rule (DS-EDLGSUB), which allows bounding types to be derived for unlinked delegations in cases where (DS-EDLG1) cannot be applied. Rules (W-MOD$\surd$), and (W-PROG) impose well-linkedness restrictions on modules and programs, respectively.

The well-linkedness judgement uses "$t.l$ virtual" and "$t.l$ undefined" to detect name clashes. These judgements were also used within DEEP--, but in DEEP-- they were part of module well-formedness. Well-linkedness also makes use of a new judgement: "$t@(G(\bar{s}))$ valid". This judgement traverses the chain of parents, and ensures that every member of every parent is well-linked within the module $G(\bar{s})$.

4.6.1 Syntax sugar and inference of annotations.

Although the new annotations are a necessary part of DEEP, they do tend to clutter the syntax. Fortunately, most of the new annotations can be easily inferred; the burden on the programmer is fairly light. In particular, we shall infer the following:

Polarity: The polarity of application and delegation can always be inferred. In the expression $t(u)^*$, $*$ is $+$ if $t \leq (\lambda^+ \leq s.\mathsf{Top})$, and $\pm$ otherwise. Similarly for delegation: in the expression $t@(u).l^*$, $*$ can be inferred by looking up l in t. The polarity of functions and declarations, however, must be explicitly specified.

Linking: The link tag on function application can be inferred in exactly the same way as polarity. For the link tag on modules, the compiler will attempt to prove that the module is well-linked. If this proof succeeds, then the module is marked as well-linked, and if it fails, then the module is marked as unlinked.

In the examples that follow, we shall only place polarity annotations on functions and declarations, and we shall only place linking annotations on functions; we assume that all other annotations are inferred. In addition, all the syntax sugar used in DEEP-- also applies to DEEP; please refer to Figure 3.2 in the previous chapter.

4.7 The DEEP partial evaluator

We have written an implemention of the DEEP calculus within Scala. Implementing the DEEP type system was fairly straightforward, and required only a few hundred lines of code. Implementing the deep partial evaluator, on the other hand, was considerably more difficult. The majority of the code in our implementation deals with partial evaluation, and most of our development time was spent fine-tuning the evaluator.

The actual implementation of the DEEP partial evaluator involves a few extra subtleties that are not part of the core calculus. We have chosen not to include these subtleties in the core calculus because they do not affect the type system. Nevertheless, they do affect the way in which the partial evaluator generates code. Since solving the DSL expression problem was one of our goals, and the DSL expression problem is a test of code generation, we describe the additional mechanisms here.

The implementation of DEEP introduces three new mechanisms to control code generation. First, DEEP uses let-expressions in order to share the results of computations, and avoid unnecessary duplication of work. Second, DEEP uses λ-lifting and closures in order to share function definitions, and avoid unnecessary duplication of code (i.e. "code bloat"). These are both standard implementation techniques that have been widely used in other languages. The third mechanism — specialization — instructs the partial evaluator to generate code instead of sharing it in certain specific cases.

4.7.1 Let expressions

The most important change involves variable substitution. The classic mathematical notation for substitution, $[x \mapsto t]u$, is easy to read and understand, but it masks an im-

$$(\lambda^*_\sigma x \le t.\ u)(s)^*_\sigma \longrightarrow \mathsf{let}\ x = s\ \mathsf{in}\ u \qquad \text{(SE-APP)}$$

$$\frac{G(\bar{t}) \rightsquigarrow \mu x\ \mathsf{extends}\ t'\ \{\overline{d}\} \quad O\ l^*\ \doteqdot u\ \in \overline{d}}{G(\bar{t})_{\surd}@(s).l^* \longrightarrow \mathsf{let}\ x = s\ \mathsf{in}\ u} \qquad \text{(SE-DLG1)}$$

$$\mathsf{let}\ x = v\ \mathsf{in}\ C[x] \longrightarrow \mathsf{let}\ x = v\ \mathsf{in}\ C[v] \qquad \text{(SE-LETVAL)}$$

$$\begin{array}{lcl} (\mathsf{let}\ x = t\ \mathsf{in}\ u)(s)^* & \longrightarrow & \mathsf{let}\ x = t\ \mathsf{in}\ u(s)^* \\ (\mathsf{let}\ x = t\ \mathsf{in}\ u)@(s).l^* & \longrightarrow & \mathsf{let}\ x = t\ \mathsf{in}\ u@(s).l^* \\ (\mathsf{let}\ x = t\ \mathsf{in}\ u)\$ & \longrightarrow & \mathsf{let}\ x = t\ \mathsf{in}\ u\$ \end{array} \qquad \text{(SE-LETCOMMUTE)}$$

$$\mathsf{let}\ x = (\mathsf{let}\ y = t\ \mathsf{in}\ u)\ \mathsf{in}\ s \longrightarrow \mathsf{let}\ y = t\ \mathsf{in}\ \ \mathsf{let}\ x = u\ \mathsf{in}\ s \qquad \text{(SE-LETASSOC)}$$

$$\frac{x \notin \mathrm{fv}(u)}{\mathsf{let}\ x = t\ \mathsf{in}\ u \longrightarrow u} \qquad \text{(SE-LETGARBAGE)}$$

$$\frac{\Gamma \vdash\ t\ \mathrm{wf} \quad \Gamma, x \equiv t \vdash\ u\ \mathrm{wf}}{\Gamma \vdash\ \mathsf{let}\ x = t\ \mathsf{in}\ u\ \mathrm{wf}}\ \text{(W-LET)} \qquad \frac{x \equiv t \in \Gamma}{\Gamma \vdash\ x \equiv t}\ \text{(DS-VAREQ)}$$

Figure 4.14: Let-expressions for partial evaluation in DEEP

portant practical issue. If x appears multiple times within u, then a straight substitution will duplicate t, and thus potentially duplicate work.

The following example shows how a simple inlining step can dramatically increase the amount of work that needs to be done, by calling foo(a) three times instead of once:

```
cube(x ≤ Int): Int inline(Top) = x*x*x;
foo(z ≤ Int): Int = ... // some expensive computation

cube(foo(a)) ⟶ foo(a)*foo(a)*foo(a)
```

The standard way to resolve this issue is to use a graph-rewriting semantics, in which x is replaced not with three separate copies of foo(a), but with three pointers to a single copy of foo(a). Graphs can be represented syntactically by let expressions. The semantics for let-expressions in DEEP are shown in figure 4.14.

The semantics shown here are essentially identical to the call-by-need λ-calculus presented in [Maraist et al., 1998]. In the untyped lambda-calculus, let-expressions need not be primitive constructs; they can be encoded as function applications. In DEEP, on the other hand, let-expressions must be primitive, because function applications do not necessarily provide enough type information. As shown in Figure 4.14,

the well-formedness rule (W-LET) uses an equivalence rather than a subtype bound on the variable, and the subtype relation is extended to handle such equivalences.

4.7.2 λ-lifting and weak-head normal forms

Although let-expressions are good for sharing the results of computations, they do not share values. Every function value (i.e. $\lambda x \leq t.\ u$) must ultimately be compiled and stored somewhere in memory. The current implementation of DEEP ensures that function values are properly shared by performing λ-lifting [Johnsson, 1985]. Every function is lifted to a top-level definition, in much the same way that modules are lifted to top-level definitions in the flattening process, described in Section 3.5.3.

Literal function values are represented as a *closures*. A closure consists of a pointer to a top-level (lifted) definition, along with a list of bindings for any variables that were defined within the lexical scope of the original un-lifted definition. Function closures are no different from module closures, which we have already included in the semantics of DEEP. A module in DEEP is written $G(\bar{t})_\sigma$; G is a "pointer" to a top level definition, while $\bar{t}$ is a list of bindings for module parameters.

Closures are weak head normal forms. Because a closure consists of a pointer to a definition which may be shared with other closures, the partial evaluator cannot step inside inside a closure to further specialize the body of the function. Partial evaluation with closures is thus restricted; it will only evaluate terms to weak head normal form.

This restriction is an important tool, because it prevents the partial evaluator from automatically specializing partial function applications (i.e. currying). Currying is heavily used in functional programming. If the partial evaluator were to indiscriminately specialize every partial application, then the result would be severe code bloat.

Every time a function is specialized or inlined, the compiler must generate the code for that function all over again. Code generation can lead to major speedups if used wisely, but it can also dramatically increase the memory footprint of a program if used unwisely. The DEEP approach is to avoid all code generation by default; the partial evaluator will only generate code when the programmer explicitly specifies that it should do so.

4.7.3 Forcing specialization

Inlining is controlled with the inline keyword, as described in section 3.6.3. However, inlining is associated with fields; the decision about whether or not to inline a func-

tion at a particular call site is not made until all of the arguments to the function are available. In order to handle currying, we introduce a similar, but slightly different mechanism, which specializes a function when only some of its arguments are available.

The specialize keyword instructs the partial evaluator to generate a specialized version of a function at a particular call site. Whereas the inline keyword is used at the point where a function is defined, the specialize keyword is used at the point where a function is called. For example:

```
M = μthis extends Top {
  final foo(x ≤ Int, y ≤ Int, z ≤ Int): Int = (x + y) + z;

  bar1 = this.foo(1,1);                    // closure
  bar2 = specialize(Top) this.foo(1,1);    // bar2 = λz ≤ Int. 2 + z;
};
```

The definition of bar1 involves a partial application of foo. This application will merely result in a closure; the partial evaluator will not attempt to produce a specialized version of foo(1,1). The definition of bar2 uses the specialize keyword to generate a new copy of foo, which has been optimized for x = 1 and y = 1. The partial evaluator generates the new copy of foo by stepping inside the closure and statically evaluating (x + y).

Like inline, specialize is parameterized by a sequence of terms. If those terms can be statically reduced to values, then specialization is done immediately, otherwise it is deferred. The expression specialize(Top) means "specialize immediately", since Top is a value.

Deferred specialization is useful for embedding a specialize command within a definition in such a way that specialization becomes automated. The following code is similar to the previous version, but it will automatically specialize foo whenever foo is called with two constant arguments. This version makes specialization work a bit more like inlining; the specialization strategy is defined at the point where foo is declared, rather than the point where foo is used:

```
M = μthis {
  final foo(x ≤ Int, y ≤ Int):
    specialize(x,y) λz ≤ Int. (: Int = (x + y) + z);

  a: Int = 1;
  bar1 = this.foo(x.a, 1);    // closure — x.a is not statically known
  bar2 = this.foo(1,   1);    // bar2 = λz ≤ Int. 2 + z;
};
```

Like inline, the specialize keyword is completely ignored at run-time; it is used only

```
Door = μthis {
  open(p ≤ Player): Bool = true;
};

MagicDoor(spell ≤ Spell, +Sup ≤ Door) = μthis extends Sup {
  override open(p ≤ Player): Bool =
    Sup@(this).open(p) && p.hasSpell(spell);
};

LockedDoor(key ≤ Key, +Sup ≤ Door) = μthis extends Sup {
  override open(p ≤ Player): Bool =
    Sup@(this).open(p) && p.hasKey(key);
};

LockedMagicDoor(key ≤ Key, spell ≤ Spell) =
  LockedDoor(key, MagicDoor(spell, Door));

MagicLockedDoor(spell ≤ Spell, key ≤ Key) =
  MagicDoor(spell, LockedDoor(key, Door));

DoubleLockedDoor(key1 ≤ Key, key2 ≤ Key) =
  LockedDoor(key1, LockedDoor(key2, Door));
```

Figure 4.15: The doors class hierarchy, in DEEP

---

for code generation at compile-time. Unlike inline, however, specialize is also completely ignored by the static type system. The inline keyword is required in order to make certain type judgements, and to break illegal cycles. If inlining fails for a particular call, then the partial evaluator is forced to generalize to a supertype. The specialize keyword has no effect on type judgements; it only affects the size and speed of the compiled code.

4.8 Examples

We illustrate the basic capabilities of the DEEP calculus through a series of examples, just as we did for DEEP-- in Chapter 3.

Our first example is the case of locked magic doors that we discussed in section 4.2.4. Figure 4.15 shows how this example can be written in DEEP; it very similar to the C++ version. The main difference between the DEEP implementation and the C++ implementation is that DEEP allows each mixin class to be modularly type-checked and separately compiled, whereas C++ does not.

The definition of DoubleLockedDoor shows a mixin composition with duplicates — a door that requires two different keys to unlock. Compositions of this form are usually impossible in languages that use linearization to implement multiple inheritance.

4.8.1 Type classes: redux

In Section 3.4.5, we showed how simple Haskell-style type classes could be encoded in DEEP--. Our encoding had one notable flaw: although we were able to establish an inheritance relationship between the type classes themselves, we could not establish an inheritance relationship between the instances of those classes. As it turns out, a proper encoding of type classes actually requires multiple inheritance. The subtype relationship between the type classes and the instances of those classes forms an inheritance diamond, as shown in figure 4.16. Since DEEP supports multiple inheritance via mixins, we can now provide a complete encoding of type classes.

Figure 4.16 shows the same Eq/Ord hierarchy that we discussed in the last chapter (Section 3.4.5). As before, Eq(T) is a module which defines functions that compare objects of type T for equality. Ord(T) then extends Eq(T) by adding additional functions for greater and less-than comparisons. The instEqInt and instOrdInt modules provide an implementation of these functions for integers.

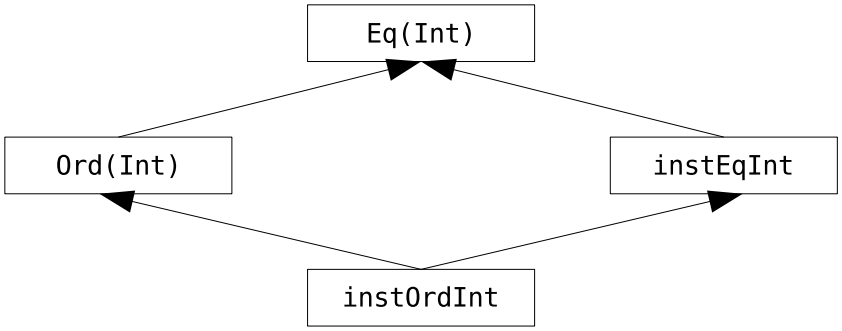
The difference between the DEEP version presented here, and the DEEP-- version presented in section 3.4.5, is that Ord(T) is defined using a mixin. OrdMix will add greater and less-than operations to any implementation of Eq(T). The instOrdInt module uses OrdMix to extend the implementation of instEqInt.

Although our use of mixins seems very different from Haskell at first glance, the encoding of type classes given here is actually faithful to the way in which type clases are implemented in Haskell. The Haskell definitions would be written as:

```
class Eq a where  ...
class (Eq a) => Ord a where ...
instance Eq Int where ...
instance Ord Int where ...
```

Notice that the Haskell definition of Ord a is prefixed by (Eq a) =>. At an abstract level, this declaration states that a type a is an instance of class Ord only if it is an instance of class Eq. In terms of the dictionary-passing semantics, however, it states that every Ord dictionary must contain an Eq dictionary. Thus, in order to construct an Ord dictionary for some type a, we must be able to obtain an Eq dictionary for a.

The OrdMix function implements the Haskell mechanism for constructing dictionar-



(above): multiple inheritance diagram for Eq and Ord.

```
Eq(T ≤ Top) = μx {
  ==(a ≤ T, b ≤ T): Bool = _;
  !=(a ≤ T, b ≤ T): Bool = !(x.==(a,b));
};

Ord(T ≤ Top) = OrdMix(T, Eq(T));
OrdMix(T ≤ Top, +Sup ≤ Eq(T)) = μx extends Sup {
  <(a ≤ T, b ≤ T): Bool = _;
  >(a ≤ T, b ≤ T): Bool = _;
  <=(a ≤ T, b ≤ T): Bool = x.==(a,b) || x.<(a,b);
  >=(a ≤ T, b ≤ T): Bool = x.==(a,b) || x.>(a,b);
};

instEqInt = μx extends Eq(Int) {
  override ==(a ≤ Int, b ≤ Int): Bool = __eqInt(a,b);
};

instOrdInt = μx extends OrdMix(Int, instEqInt) {
  override <(a ≤ Int, b ≤ Int): Bool = __ltInt(a,b);
  override >(a ≤ Int, b ≤ Int): Bool = __gtInt(a,b);
};

max(T ≤ Top, ord ≤ Ord(T), a ≤ T, b ≤ T): Bool =
  if (ord.>(a,b)) then a else b;

myMaxValue = max(Int, instOrdInt, 3, 4);
```

Figure 4.16: Example of type classes in DEEP

---

ies. Because every Ord dictionary must contain an Eq dictionary, we pass the necessary Eq dictionary as a formal parameter to Ord. The subtype rules for mixin composition ensure that Ord(T) ≤ Eq(T), and that instEqInt ≤ instOrdInt, as one would expect. (See Section 4.3.5.)

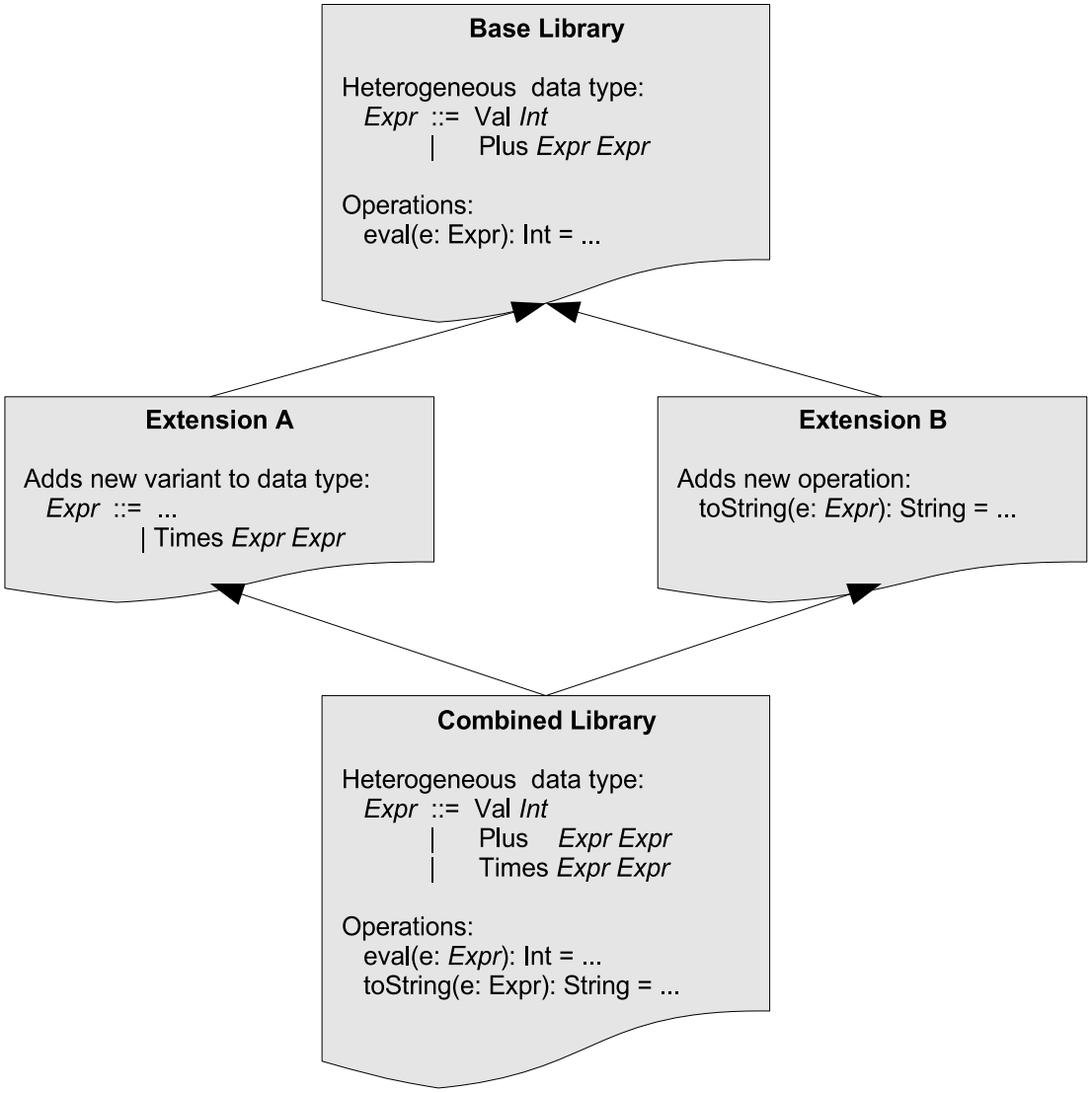


Figure 4.17: The Expression Problem

4.8.2 The Expression Problem: Introduction

The "expression problem" is one of the main examples which motivated the design of the DEEP calculus. The problem itself is discussed in more detail in Chapter 1; the reader may wish to refer back to that section for a more detailed description.

The expression problem concerns the design of an interpreter which can handle simple arithmetic expressions. The interpreter consists of data type for abstract syntax trees (ASTs), and a set of operations on those ASTs. An abstract syntax tree consists of several different variants of node which make up the leaves and branches of the tree. The terminals (leaves) of the tree are integer literals, while non-terminals may

include addition, multiplication, division, and so on. Operations on ASTs may include evaluation, pretty printing, optimization passes, type-checking, and so on.

A solution to the expression problem is a language, language mechanism, or programming technique which satisfies the following criteria, which are illustrated in Figure 4.17:

- It must extend the data type with new variants, without altering the source code of the original base module.
- It must add new operations, without altering the source code of the original base module.
- It must combine two different extensions together.
- It must be possible to type-check and compile each extension separately.

In essence, the expression problem is a code factorization problem. It tests the ability of a language to factor functionality into modules which can be separately compiled. As described in Section 1.2, neither object-oriented nor functional languages provide a convenient language mechanism for dealing with this scenario.

4.8.3 The Expression Problem: OO Encoding

Figure 4.18 shows how a solution to the expression problem can be encoded in DEEP in an object-oriented style, following the interpreter design pattern [Gamma et al., 1995]. Base is a base module, which defines the Expr class, along with two variants: Lit for integer literals, and Plus for addition. It also defines one operation: eval.

The Print module adds a new method toString, which will convert an expression to a string. The Multiply module adds a new variant: Times for multiplication.

The MultPrint module mixes Print and Multiply together. Since the Print module has no knowledge of the Times variant, it does not define a toString method for Times nodes. Likewise, the Multiply module does not have any knowledge of Print, and thus does not define pretty printing either. When we mix the two modules together we must define the missing piece – an implementation of toString for Times nodes.

The presence of a "missing piece" is a consequence of the fact Print and Multiply are not entirely orthogonal extensions. Such non-orthogonality is common in software product lines and feature-oriented software development [Batory et al., 2003] [Pulvermuellern et al., 2002] [Liu et al., 2005]. Although the goal of feature-oriented programming is to factor a piece of software into a set of features which are as inde-

```
Base = μx {
  Expr: μthis {
    eval: Int = _;
  };

  Lit(i ≤ Int): μthis extends x.Expr {
    override eval: Int = i;
  };

  Plus(a ≤ x.Expr, b ≤ x.Expr): μthis extends x.Expr {
    override eval: Int = a.eval + b.eval;
  };

  twoPlusTwo = x.Plus(x.Lit(2), x.Lit(2));
};

Print(+Sup ≤ Base) = μx extends Sup {
  override Expr: μthis extends Sup@(x).Expr {
    toString: String = _;
  };

  override Lit(i ≤ Int): μthis extends Sup@(x).Lit(i) {
    override toString: String = int2String(i);
  };

  override Plus(a ≤ x.Expr, b ≤ x.Expr): μthis extends Sup@(x).Plus(a,b) {
    override toString: String = "(" + a.toString ++ "+" ++ b.toString ++ ")";
  };
};

Multiply(+Sup ≤ Base) = μx extends Sup {
  Times(a ≤ x.Expr, b ≤ x.Expr): μthis extends x.Expr {
    override eval: Int = a.eval * b.eval;
  };

  Square(a ≤ x.Expr) = x.Times(a, a);
  fourSquared = x.Square(x.twoPlusTwo);
};

MultPrint = μx extends Multiply(Print(Base)) {
  override Times(a ≤ x.Expr, b ≤ x.Expr):
    μthis extends Multiply(Print(Base))@(x).Times(a,b)
  {
    override toString: String = "(" + a.toString ++ "*" ++ b.toString ++ ")";
  };
};

answer = MultPrint.fourSquared.toString ++ " = " ++
         int2String(MultPrint.fourSquared.eval);
```

Figure 4.18: An OO-style solution to the expression problem in DEEP

```
data Expr = Lit Int
          | Plus Expr Expr
          | Times Expr Expr

eval :: Expr -> Int
eval (Lit i)     = i
eval (Plus a b)  = (eval a) + (eval b)
eval (Times a b) = (eval a) * (eval b)

toString :: Expr -> String
toString (Lit i)     = (show i)
toString (Plus a b)  = "(" ++ (toString a) ++ "+" ++ (toString b) ++ ")"
toString (Times a b) = "(" ++ (toString a) ++ "*" ++ (toString b) ++ ")"
```

Figure 4.19: A simple interpreter in Haskell

---

pendent as possible, the reality is that features may interact in various ways when they are composed together. Feature interactions are not a language or a typing issue, they are a software design and modeling issue. At the language level, we need only ensure that when non-orthogonal features do interact, it is possible apply a "patch" that glues things together in an appropriate way, as shown in this example.

To demonstrate the robustness of our solution, Figure 4.18 implements some additional methods to construct an example AST. The fourSquared method is defined in the Multiply module, while twoPlustwo is defined in the Base module. Neither of these modules defines toString. Yet when we invoke fourSquared from the MultPrint module, the magic of late binding ensures that the resulting AST can be both evaluated and pretty-printed.

4.8.4 The Expression Problem: FP encoding

Although a number of solutions to the expression problem have been proposed in the literature, such solutions often require the use of a particular programming style or design pattern [Krishnamurthi et al., 1998] [Torgersen, 2004]. Achieving extensibility in this way is not ideal, because there is no one programming style or design pattern that is appropriate for every problem domain. We believe that extensiblity should be built into the language in such a way that it does not matter which programming style or design pattern is being used.

Virtual types and late binding are general-purpose techniques, which are flexible

enough to allow many different programming styles. We illustrate this flexibility by showing a second solution to the expression problem, one which uses a functional programming style based on variant data types and pattern matching.

Figure 4.19 shows a simple arithmetic interpreter written in Haskell. (It is the same as Figure 1.2, except that we haved added a Times variant.) We show how this definition can be encoded in DEEP, and then factored into the same modules that we used in the OO-style solution, where Times, toString, and toString (Times a b) are all defined different modules.

Figure 4.20 shows our FP-style solution. A minor complication is the fact that DEEP does not provide native support for variant data types and pattern matching. As is standard practice, we translate pattern matching into more primitive constructs in the core calculus. This translation is somewhat heavy, but would be done automatically in a real implementation. For clarity, we have put Haskell pseudo-code in comments.

Our implementation of pattern matching is somewhat similar to the visitor design pattern [Gamma et al., 1995]. The Expr class defines a single method match, which requires an argument of type ExprCases. ExprCases is a record which defines one method for each particular pattern matching case. It also provides a default method, which is a catch-all case for situations in which the caller of match does not provide a full spanning set of patterns.

Note that in the FP-style solution, the virtual type is ExprCases, not Expr. In order to add a new variant, we must add a new method to ExprCases. We must also extend every function which uses pattern matching so that it recognizes the new variant. We extend a function by using the default case to forward any unhandled patterns to super, as shown in the MultPrint module.

4.8.5 The DSL Expression Problem

As we discussed in Chapter 1, the DSL expression problem is the main motivating example for this thesis. It serves as an "acid test" for the DEEP calculus, because it combines two different sub-problems, each of which is already quite difficult to solve, into one example. The first part of the DSL expression problem is the expression problem, which tests the ability of a language to factor out behavior into separate modules. The second part is the tag-elimination problem, which tests the ability of a partial evaluator to eliminate certain computations in a statically typed language.

Both sub-problems have been widely studied in the literature, but they have been

```
Base = μx {
  // data Expr = Lit Int
  //           | Plus Expr Expr

  ExprCases(RT ≤ Top): μy {
    lit (i ≤ Int): RT                = y.default;
    plus(a ≤ x.Expr, b ≤ x.Expr): RT = y.default;
    default: RT = _;
  };

  Expr = μy {
    match(RT ≤ Top, cases ≤ x.ExprCases(RT)): RT = _;
  };

  Lit(i ≤ Int) = μy extends x.Expr {
    override match(RT ≤ Top, cases ≤ x.ExprCases(RT)): RT = cases.lit(i);
  };

  Plus(a ≤ x.Expr, b ≤ x.Expr) = μy extends x.Expr {
    override match(RT ≤ Top, cases ≤ x.ExprCases(RT)): RT = cases.plus(a,b);
  };

  // eval :: Expr -> Int
  // eval (Lit i)    = i
  // eval (Plus a b) = (eval a) + (eval b)

  eval(e ≤ x.Expr): Int = e.match(Int, μy extends x.ExprCases(Int) {
    override lit(i ≤ Int): Int                = i;
    override plus(a ≤ x.Expr, b ≤ x.Expr): Int = eval(a) + eval(b);
  });
};

Print(+Sup ≤ Base) = μx extends Sup {
  // toString :: Expr -> String
  // toString (Lit i)    = int2String(i)
  // toString (Plus a b) = "(" ++ (toString a) ++ "+" ++ (toString b) ++ ")"

  toString(e ≤ x.Expr): String =
    e.match(String, μy extends x.ExprCases(String) {
      override lit(i ≤ Int): String              = int2String(i);
      override plus(a ≤ x.Expr, b ≤ x.Expr): Int =
        "(" ++ toString(a) ++ "+" ++ toString(b) ++ ")";
    });
};
```

(continued on next page)

Figure 4.20: An FP-style solution of the expression problem in DEEP

---

studied by different research communities. The two sub-problems are similar because their solution requires a very sophisticated type system. Although other type systems have been developed which are capable of handling either one sub-problem or the

(continued from previous page)

```
Multiply(+Sup ≤ Base) = μx extends Sup {
  // data Expr = super.Expr
  //           | Times Expr Expr

  override ExprCases: μy extends Sup@(x).ExprCases {
    times(a ≤ x.Expr, b ≤ x.Expr): RT = y.default;
  };

  Times(a ≤ x.Expr, b ≤ x.Expr) = μy extends x.Expr {
    override match(RT ≤ Top, cases ≤ x.ExprCases(RT)): RT = cases.times(a,b);
  };

  // eval Times(a,b) = eval(a) * eval(b)
  // eval e          = super.eval(e)

  override eval(e ≤ x.Expr): Int = e.match(Int, μy extends x.ExprCases(Int) {
    override default: Int                       = Sup@(x).eval(e);
    override times(a ≤ x.Expr, b ≤ x.Expr): Int = x.eval(a) * x.eval(b);
  });
};

MultPrint = μx extends Multiply(Print(Base)) {
  // toString Times(a,b) = "(" ++ (toString a) ++ "*" ++ (toString b) ++ ")"
  // toString e          = super.toString(e)

  override toString(e ≤ x.Expr): String =
    e.match(String, μy extends x.ExprCases(String) {
      override default: String = Multiply(Print(Base))@(x).toString(e);
      override times(a ≤ x.Expr, b ≤ x.Expr): Int =
        "(" ++ toString(a) ++ "*" ++ toString(b) ++ ")";
    });
};
```

---

other, there is, to our knowledge, no programming language or formal calculus other than DEEP which is capable of solving both sub-problems simultaneously.

4.8.6 The tag-elimination problem

Partial evaluation has the potential to automatically transform an interpreter into a compiler. Interpreters are slow because they traverse the abstract syntax tree (AST) of a program at run-time, branching on each node in order to decide what to do. Portions of the AST which are executed multiple times, such as loops or recursive functions, must be traversed multiple times. Compilers achieve a major speedup by traversing the AST once, and translating it to some implementation language that can be efficiently executed. This speedup is possible because the AST (i.e. the source code) is typically

known at compile-time, and remains constant.

Since the AST is statically known, partial evaluation can also be used to traverse the AST at compile-time. Partially evaluating an interpreter with respect to the AST should eliminate all of the interpretive overhead, yielding a residual program that is just as fast as compiled code [Jones et al., 1993]. This technique is an excellent way to efficiently implement domain-specific languages (DSLs), because it is much easier to write an interpreter for a DSL than it is to write a compiler.

The tag elimination problem describes a situation where partial evaluation is unable to eliminate all of the interpretive overhead [Hughes, 1996] [Taha and Makholm, 2000] [Pašalić et al., 2002] [Carette et al., 2007]. It arises when an intepreter for an untyped or dynamically typed language is written in a statically typed language. Section 1.3.3 provides a detailed example of the tag-elimination problem; the reader may wish to refer back to that section before moving on.

In a dynamically typed language, the type of an expression is not known until run-time. The interpreter for such a language must therefore tag each piece of data with its run-time type. Every operation on data must check the tag to ensure that the data has the correct type, remove the tag to get the actual data, and then add a tag to the result of the computation. For example, in order to perform an addition, the interpreter must verify that the arguments are integers, extract the integer data, add the integers, and then tag the result as an integer.

For simple operations like addition, the extra work required to check and manipulate tags can easily slow down the code by an order of magnitude. Unfortunately, the tags and tagging operations are difficult to eliminate using standard techniques. Whereas the AST is statically known, the type tags are not, so partial evaluation is unable to remove them.

A solution to the tag-elimination problem has the following three elements:

- A statically typed language L_1.
- A partial evaluator for L_1.
- An interpreter written in L_1, which interprets another language L_2.

In order to solve the tag-elimination problem, the partial evaluator must be able to translate L_2 to L_1 and completely remove all interpretive overhead, *including type tags*. L_2 is ususally chosen to be a subset of L_1, so that the accuracy of the solution can be easily verified. For our solution, L_1 is DEEP, and L_2 will be the λ-calculus.

4.8.7 Type-indexed variant data types

There is no way to eliminate dynamic type checks by tweaking the partial evaluator, because the fundamental problem is the fact that the interpreted language (L_2) is untyped. In an untyped or a dynamically typed language, the types (and thus the tags) are not statically known. The only way to eliminate tags is to statically determine the types, which we shall do by adding a type system to our interpreted language.

Instead of writing an interpreter for the untyped λ-calculus, we will write an interpreter for the simply-typed λ-calculus. The new interpreter eliminates the universal data type shown in Chapter 1, Figure 1.5, which was the cause of our trouble. Instead of using a universal (i.e. tagged) data type, we index each variant of Expr with the type of value that will be returned when the expression is evaluated. Lit and Plus nodes are indexed with type Int, Fun nodes are indexed with function types, and so on.

These type indexes differ from ordinary type tags because they are generated at compile-time, as part of the AST. When the interpeter traverses the AST, the indexes will be eliminated. We have defined our data types such that it is only possible to construct ASTs that are well typed. For example, the both of the arguments to Plus must be integer expressions; it is not possible to construct a Plus node in any other way.

Because there is no universal data type, there is no need to perform dynamic type checks, and the partial evaluator will generate code which contains no interpretive overhead.

4.8.8 Covariant tuples

Type judgements for the simply-typed λ-calculus have the form $\Gamma \vdash t : T$. That is to say, a term t can only be judged to have a type T within a particular context Γ. In order to handle contexts, we must be able to do three things:

1. Represent a context Γ, which lists the types of free variables.
2. Represent an environment env, which lists the values of free variables.
3. Represent the fact that a particular environment matches a particular typing context.

We handle these three things with *covariant tuples*. A typing context is represented as a tuple of types, e.g. (Int, Int → Int, Int), and an environment is a tuple of objects, e.g. (3, $\lambda x \leq$ Int. x, 5). An environment matches its context if the environment is a subtype

```
// tuples for type contexts and environments
Tuple = μx {
  getElem+(n ≤ Int): Top;
  length: Int = _;
};

TupNil = μx extends Tuple {
  getElem+(n ≤ Int): error;
  length: Int inline(Top) = 0;
};

TupCons(+head: Object, +tail: Context) = μx extends Tuple {
  getElem+(n ≤ Int): if (n == 0) then head else tail.getElem(n−1);
  length: Int inline(Top) = tail.length + 1;
};
```

Figure 4.21: Tuples in DEEP

---

of its context.

Figure 4.21 shows how tuples can be encoded in DEEP. Tuples are very similar to lists, except that the head and tail arguments are covariant. (We refer to them as "tuples" rather than "lists" by analogy with Haskell. In Haskell, a list holds multiple objects of the same type, whereas a tuple holds multiple objects of differing types. DEEP does not necessarily require such a distinction; tuples and lists are pretty much the same thing.)

The getElem method will project the n^{th} element in a tuple. Since getElem is also covariant, we know that $t \leq u$ implies $t.\mathsf{getElem}(n) \leq u.\mathsf{getElem}(n)$. Note also that getElem acts like a dependently-typed function if the tuple contains types.

Since DEEP is intended as a language for practical programming, our definition of tuples takes a shortcut. A truly safe definition should constrain the argument of getElem to be a natural number which is smaller than the length of the tuple. This sort of thing is certainly possible, and is routinely done in dependent type systems, but it complicates both the definition and use of tuples. We cheat and define getElem over integers, and then throw an error if the argument is out of range. The error keyword is defined in the current implementation of DEEP. It is a subtype of all terms, and will throw a run-time error when evaluated.

As a side comment, tuples neatly illustrate one of the practical benefits of unifying types and objects. Our representation of contexts and environments is particularly

simple because we can use the same data structure to store both types and objects.

4.8.9 A well-typed interpreter – OO style

Figure 4.22 shows an interpeter for the simply-typed λ-calculus. Each node of the AST is parameterized by the type of the node, and the type context in which that type has been assigned. An expression must be evaluated within an environment that matches its type context.

Notice that the definition of Fun.eval makes use of the specialize keyword. The eval method returns a function, and makes a recursive call to eval within the body of that function. Ordinarily, functions are treated as closures, which means that the partial evaluator will only reduce them to weak-head normal form; it will not step inside the function to evaluate the body. The specialize keyword ensures that it does evaluate the body; every invocation of Fun.eval will generate a new function.

The presence of type and context tags obviously makes the task of constructing an AST far more tedious in the typed version than it is in the untyped version. However, in a more robust implementation of DEEP, these arguments would be declared as *implicit arguments* [Lewis et al., 2000] [Luo and Pollack, 1992]. Implicit arguments can be inferred from the calling context in much the same way that type arguments to polymorphic functions are inferred in Haskell and Generic Java. Using implicit arguments, the definition of timesTwoE would look like: timesTwoE = x.Fun(x.Plus(x.Var(0), x.Var(0))), which is comparable to the Haskell code in Figure 1.5. The burden on a programmer who is using the interpreter is thus not necessarily as high is it appears to be in this example.

4.8.10 Pretty printing

All of the classes declared in BaseMod are virtual classes. As a result, they can be extended with new methods, just like any other virtual class. Our use of type indexes does not affect their extensibility in any way.

Figure 4.23 defines a mixin which adds pretty printing to BaseMod; the basic technique is the same as for the vanilla expression problem. Adding a new class, and mixing two extensions together is also exactly the same as in the vanilla expression problem; we thus omit the details.

```
// base module for the simply typed lambda calculus
BaseMod = μx {
  Expr(Ctx ≤ Tuple, Typ ≤ Top): μy {
    eval(env ≤ Ctx): Typ = _;
  };

  Lit(Ctx ≤ Tuple, n ≤ Int): μy extends x.Expr(Ctx, Int) {
    override eval(env ≤ Ctx): Int inline(Top) = n;
  };

  Var(Ctx ≤ Tuple, idx ≤ Int): μy extends x.Expr(Ctx, Ctx.getElem(idx)) {
    override eval(env ≤ Ctx): Ctx.getElem(idx) inline(Top) =
      env.getElem(idx);
  };

  Fun(Ctx ≤ Tuple, InT ≤ Top, OutT ≤ Top,
      body ≤ Expr(TupCons(InT, Ctx), OutT)):
    μy extends x.Expr(Ctx, InT → OutT)
  {
    override eval(env ≤ Ctx): InT → OutT inline(Top) =
      specialize(env) λ(z ≤ InT): OutT = body.eval(TupCons(z, env));
  };

  Plus(Ctx ≤ Tuple, l ≤ Expr(Ctx, Int), r ≤ Expr(Ctx, Int)):
    μy extends x.Expr(Ctx, Int)
  {
    override eval(env ≤ Ctx): Int inline(Top) =
      l.eval(env) + r.eval(env);
  };

  Apply(Ctx ≤ Tuple, InT ≤ Top, OutT ≤ Top,
        fun ≤ Expr(Ctx, InT → OutT),
        arg ≤ Expr(Ctx, InT)):
    μy extends x.Expr(Ctx, OutT)
  {
    override eval(env ≤ Ctx): OutT inline(Top) =
      (fun.eval(env))(arg.eval(env));
  };

  // timesTwo = λx ≤ Int. x + x
  timesTwoE = x.Fun(TupNil, Int, Int,
    x.Plus(TupCons(Int, TupNil), x.Var(TupCons(Int, TupNil), 0),
                                 x.Var(TupCons(Int, TupNil), 0)));
};

// partially evaluates to λx ≤ Int. x + x
timesTwo = BaseMod.timesTwoE.eval(TupNil)
```

Figure 4.22: Tag elimination in DEEP

```
// base module for the simply typed lambda calculus
PrintMod(Sup ≤ BaseMod) = μx extends Sup {
  override Expr(Ctx ≤ Tuple, Typ ≤ Top): μy extends Sup@(x).Expr(Ctx, Typ) {
    toString: String = _;
  };

  Lit(Ctx ≤ Tuple, n ≤ Int): μy extends Sup@(x).Lit(Ctx, n) {
    override toString: String = int2String(n);
  };

  Var(Ctx ≤ Tuple, idx ≤ Int): μy extends Sup@(x).Var(Ctx, idx) {
    override toString: String = "x" ++ int2String(Ctx.length − idx);
  };

  Fun(Ctx ≤ Tuple, InT ≤ Top, OutT ≤ Top,
      body ≤ Expr(TupCons(InT, Ctx), OutT)):
    μy extends Sup@(x).Fun(Ctx, InT, OutT, body)
  {
    override toString: String =
      "(\\x" ++ int2String(Ctx.length+1) ++ ". " ++ body.toString ++ ")";
  };

  Plus(Ctx ≤ Tuple, l ≤ Expr(Ctx, Int), r ≤ Expr(Ctx, Int)):
    μy extends Sup@(x).Plus(Ctx, l, r)
  {
    override toString: String =
      "(" ++ l.toString ++ " + " ++ r.toString ++ ")";
  };

  Apply(Ctx ≤ Tuple, InT ≤ Top, OutT ≤ Top,
        fun ≤ Expr(Ctx, InT → OutT),
        arg ≤ Expr(Ctx, InT)):
    μy extends Sup@(x).Apply(Ctx, InT, OutT, fun, arg)
  {
    override toString: String = fun.toString ++ "(" ++ arg.toString ++ ")";
  };
};

InterpMod = PrintMod(BaseMod);

// evaluates to "(\x1. (x1 + x1))"
myStr = InterpMod.timesTwoE.toString;
```

Figure 4.23: Pretty printing

4.8.11 A well-typed interpreter – FP style

Since we have presented a solution to the DSL expression problem in an object-oriented style, it would be natural to present a solution in a functional programming (FP) style as well, the same as we did for the vanilla expression problem. Unfortunately, although it is possible to write an FP-style solution in DEEP, the encoding becomes excessively heavy.

Our FP-style solution to the vanilla expression problem used standard variant data types, which can be easily encoded in the core calculus. However, the DSL expression problem requires a more advanced version of variant data types called *generalized algebraic data types* (GADTs) [Peyton Jones et al., 2006], which are also known as *guarded recursive datatypes* [Xi et al., 2003] or *first-class phantom types* [Cheney and Hinze, 2003].

GADTs make pattern matching much more complicated, because matching a particular pattern may cause the types of other variables to be refined. For example, the following code is written in a hypothetical version of DEEP with pattern matching:

```
Expr(Ctx ≤ Tuple, T ≤ Top): μx { ... };
Lit (Ctx ≤ Tuple, n ≤ Int): μx extends x.Expr(Ctx, Int) { ... };
Var (Ctx ≤ Tuple, i ≤ Int): μy extends x.Expr(Ctx, Ctx.getElem(i))...

...
eval(Ctx ≤ Tuple, T ≤ Top, expr ≤ x.Expr(Ctx, T), env ≤ Ctx): T =
  case expr of {
    Lit(n) -> n;                // must show that T ≡ Int
    Var(i) -> env.getElem(i);   // must show that T ≡ Ctx.getElem(i)
    ...
  };
```

Every variant of Expr has a different type tag associated with it; e.g. the tag on Lit is Int. The eval function is defined over all variants, so within the body of eval, expr is a subtype of Expr(Ctx, T). However, once we have matched expr against a particular pattern, our knowlege of its type changes. If expr is an instance of Lit, then we know that T ≡ Int. If expr is an instance of Var, then we know that T ≡ Ctx.getElem(idx), and so on. This additional type information is necessary in order to prove that the body of eval is well-formed.

In principle, such an encoding can be done without any actual extension to the core calculus. Under the Curry-Howard interpretation of types and objects, a proof that T ≡ U is a function that will cast any term of type T to a term of type U; details for the dependently typed case can be found in [McBride and McKinna, 2004].

In practice, modeling type casts as ordinary functions is undesirable because it would introduce too much overhead. This overhead is especially important if we are attempting to test the performance of the partial evaluator. A type cast is a special kind of function; it's a function that has no computational effect, and thus can be erased at compile-time. An extension to System F which supports such casts is described in [Sulzmann et al., 2007].

Dealing with type casts in DEEP is likely to be much more complex than System F, because DEEP supports both higher-order types and subtyping. Thus, not only is the notion of type equality more complex (because of β-reduction), a complete solution would ideally include subtype coercions as well. Because of these complications, implementing GADTs within DEEP remains firmly in the category of "future work".

4.9 Future work

Like any language, DEEP is still evolving. As our difficulty in encoding GADTs demonstrates, there are still areas where either the DEEP calculus, or the DEEP type system, is not as powerful as we would like it to be. This section gives an overview of areas where we think the DEEP calculus could be improved, along with a few ideas on how to improve it.

4.9.1 Linking

If we had to choose one piece of theory in this thesis that was most in need of improvement, it would be the DEEP linking mechanism. As far as we know, there is nothing technically wrong with the linking mechanism, but there are several reasons, both technical and aesthetic, for wishing to improve it.

First, the DEEP linking mechanism requires extensions to the syntax of DEEP which are somewhat unintuitive. Second, linking is restricted to static compositions; we would like it to cover dynamic composition as well. Third, the well-linkedness judgement interacts with both subtyping and well-formedness in non-trivial ways. Since the meta-theory of System $\lambda_{\lhd}$ is already intractable, adding yet another judgement with subtle dependencies seems unwise. And finally, although linking is performed by the partial evaluator, the actual reduction steps are hidden; they do not appear as proper reduction rules. This section covers some possible ways of addressing these shortcomings.

4.9.1.1 Linking as a computation

The basic idea behind linking is sound. It is very difficult to detect name clashes before composition, so we detect them after composition instead. We initially mark a term as being *unlinked*, meaning that it may contain composition errors, and then use the partial evaluator to reduce it to a well-linked term, which is verifiably error-free. Reducing the initial term will perform the necessary compositions, thus revealing any name-clashes.

Unfortunately, the way in which partial evaluation is used is rather unintuitive. Linking in DEEP is done by the following rule. The rule is written as an equivalence in the declarative formulation of subtyping, but would be a reduction rule in the algorithmic formulation:

$$\frac{\Gamma \vdash\ G(\bar{t})\ \text{wlk}}{\Gamma \vdash\ G(\bar{t})_{?} \equiv G(\bar{t})_{\surd}}(\text{DS-ELINK})$$

The well-linkedness judgement will step inside $G(\bar{t})$ and partially evaluate all parents and slots to a well-linked form. However, once everything has been evaluated, the result of that evaluation is discarded. The linking step doesn't actually reduce the module, it only changes the linking annotation: ? switches to $\surd$.

Intuitively, we expect linking to be a computation, not just a safety check. In other words, the result of linking should be a program in which various variables and/or slots have been bound to their final definitions. Instead, what DEEP gives us is the original program, which has been marked as "safe to link".

There are two fundamental difficulties that prevent us from modeling linking as a computation. The first difficulty is a result of the μ-lifting process described in Section 3.5.3, and the second is a result of late-binding.

4.9.1.2 Linking inside a μ-abstraction

As described in Section 3.5.3, all μ-abstractions in DEEP must be lifted to top-level definitions. Modules are represented using the syntax $G(\bar{t})$, instead of μx extends u $\{\overline{d}\}$. Consequently, only the parameters $\bar{t}$ can be reduced; it is not possible for ordinary reduction rules to step inside a μ-abstraction and reduce the parent u or the slots $\overline{d}$.

If we were using the surface syntax of DEEP (similar to the surface syntax of DEEP--), then we would be able to write reduction rules like the following:

$$\mu x \text{ extends } (\mu x \text{ extends } t\ \{\overline{c}\})\ \{\overline{d}\} \longrightarrow \mu x \text{ extends } t\ \{\overline{e}\}$$
$$\text{where } \mathrm{dom}(\overline{e}) = dom(\overline{c}) \cup dom(\overline{d})$$
$$\text{and } e_\ell = \begin{cases} c_\ell & \text{if } \ell \in \mathrm{dom}(\overline{c}) \text{ and } \ell \notin \mathrm{dom}(\overline{d}) \\ d_\ell & \text{if } \ell \in \mathrm{dom}(\overline{d}) \text{ and } \ell \notin \mathrm{dom}(\overline{c}) \\ c_\ell \oplus d_\ell & \text{if } \ell \in \mathrm{dom}(\overline{c}) \cap \mathrm{dom}(\overline{d}) \end{cases}$$
$$\text{def } l : t \ \oplus\ \text{override } l : u \ =\ \text{def } l : u$$
$$\text{def } l : t \ \oplus\ \text{override } l = u \ =\ \text{def } l = u$$

This rule looks much more like the composition rules found in other theories of linking, like Cardell's linking system [Cardelli, 1997] or Wells' M-calculus [Wells and Vestergaard, 2000]. The rule merges the definitions of two modules together, where overriding definitions replace earlier ones with the same name.

Notice that it is not possible for a final binding to be overridden, nor is it possible to add a new slot with the same name as a previous slot: $\text{def } l : t \oplus \text{def } l : u$ is undefined. It therefore not possible to merge two modules together if there are any name clashes. In the presence of a name clash, the above reduction rule will fail.

By treating linking as a computation, rather than a safety check, we can determine well-linkedness in a much simpler way. A well-linked module is one which can be reduced to μx extends Top $\{\overline{d}\}$. Since name clashes are caused by inheritance, a module which inherits from Top is guaranteed to be free of any immediate name clashes, although clashes may exist in nested modules.

4.9.1.3 Linking of nested modules

In order to detect name clashes in nested modules, the mechanism of *deep-linking* states that a module is only well-linked if all its slots are well-linked (see section 4.5.7.2). Switching to the surface syntax of DEEP would allow us to reduce the slots of a μ-abstraction, but this alone is not sufficient for virtual inheritance. Consider the following example:

```
M = μx extends Top {
  A: μy extends Top { ... };
  B: μy extends x.A { ... };
}
```

We cannot reduce B to a well-linked form because DEEP uses late binding. Inside a μ-abstraction, x is a free variable, and x.A cannot be further reduced. The well-

linkedness judgement circumvents this issue by using early-binding instead; it evaluates B under the assumption that $x \equiv \mathsf{M}$.

The assumption that $x \equiv \mathsf{M}$ does not hold in subtypes of M, so the well-linkedness judgement will have to redo all of the linking steps in each derived module. This is the reason why the linking judgement discards all of the work done by the partial evaluator.

4.9.1.4 Circular definitions

Late binding affects more than just linking. It also makes certain common optimizations much more difficult. In particular, it makes it hard to optimize recursive functions and circular data structures. Consider the following module:

```
M = μx extends Top {
  tailsum(lst ≤ List(Int), total ≤ Int): Int =
    case lst of {
      Nil         -> total;
      Cons(y, ys) -> x.tailsum(ys, total + y);   // tail call
    };
};
```

This code is attempting to define a tail-recursive function. In a language like Haskell, the recursion would be optimized away, replaced by a simple loop in the compiled code. DEEP cannot transform the recursion into a loop, because x is a free variable within within the body of M, and the definition of x.tailsum is not statically known. We cannot safely eliminate the call to x.tailsum because it could be overridden in a derived module.

Nevertheless, note what happens when we project tailsum from M:

```
M.tailsum = λ(lst ≤ List(Int), total ≤ Int): Int =
    case lst of {
      Nil         -> total;
      Cons(y, ys) -> M.tailsum(ys, total + y);   // recursive function
    };
};
```

If we solve for M.tailsum, we obtain a more conventional recursive definition. Projecting a slot from a module essentially takes the fixpoint of the module, thus allowing optimizations that were not previously possible.

Unfortunately, exploiting this behavior is difficult. The compiler must solve the fixpoint equations for expressions such as M.tailsum, rather than variables as is usually

done. Even worse, such expressions may be scattered throughout the code. There is no single location in the code where the fixpoint of M is actually computed, so each recursive slot must be dealt with separately.

4.9.1.5 Final modules: a potential solution

Java allows a class to be declared final, which means that it is not allowed to have any subclasses. Applying the same idea to DEEP, we could define a *final module* as a module that cannot be extended.

If x is the self-variable of a final module M, then x need not be regarded as a free variable. It would be safe to permanently assume that $x \equiv M$. Instead of late binding, the module would use early binding, thus allowing linking steps or circular definitions to be resolved.

One complication with this strategy is that only final modules would be well-linked. This is not as serious a restriction as it sounds, since for any non-final module N, a final version could be created as μx final extends N $\{\}$. However, it would still be necessary to deal with non-final modules in some way. One possibility would be to use a universe judgement to label non-final modules as "abstract types" and thus prevent them from being used in concrete computations (see Section 2.3.4).

4.9.1.6 Monadic linking

The linking system of DEEP currently uses various syntactic annotations to track whether terms are well-linked. Such annotations are a rather ad-hoc solution. A more principled way of approaching the problem would be to use *monads* instead, which are a highly developed and well-understood way of dealing with side-effects of various kinds [Wadler, 1992].

An unlinked term is simply one for which evaluation can fail, and failure is one of the simplest side-effects that can be defined. Moreover, failure is a side-effect that can happen at either compile-time or run-time, so a monadic linking system could potentially perform both static and dynamic mixin composition.

The trouble with using monads is that once side-effects are added to the core calculus, they have a tendency to propogate throughout computations. In DEEP, this is especially a problem because any term can act as a type, and we would thus have to define a subtype relation over expressions that include side effects.

We abandoned our initial attempt to define linking within a monad due to these

complications. However, the linking model as currently defined is not necessarily any simpler, so this is an issue well worth revisiting.

4.9.1.7 Linking: summary of future work

We would like to eliminate the well-linkedness judgement from DEEP. The current definition of DEEP uses partial evaluation to prove that modules are well-linked. However, once the well-linkedness check has completed, the results of all those computations are discarded. This mechanism is complex, unintuitive, and it fails to perform certain valuable optimizations.

Ideally, linking should be done as part of ordinary reduction, with link errors handled by a monad. This would not only be simpler and more efficient, but it would eliminate one of the current restrictions in DEEP: that all mixin compositions must be static. In order to achieve this goal, we will need to resolve two separate tensions within the current caclulus:

(1) There is a tension between the formal syntax of DEEP, in which modules are lifted to top-level definitions, and the surface syntax of DEEP, in which they are not. Module equality and subtyping require μ-lifting in order to be sound. Linking, however, requires reductions that can step inside μ-abstractions.

(2) There is a tension between early binding and late binding. Late binding is required for modules to be extensible, whereas early binding is required in order for modules to be well-linked.

Resolving these tensions properly is a very subtle problem. The current DEEP linking system is not elegant, but at least it works, and can serve as a stopgap measure until a more suitable system can be designed.

4.9.2 Dynamic type checks

One of the most valuable extensions to the DEEP type system would be the addition of dynamic type information. Dynamic typing is usually regarded as a substitute for static typing. Languages like Perl, Python, Smalltalk, and Lisp are described as "dynamically typed" because they do not have a static type system.

However, dynamic typing can also be combined with static typing, especially in languages which support subtyping. Java provides a static type system, but it also maintains dynamic type information on all objects. The Java virtual machine uses dynamic type information to perform "safe" type-casts; such casts are needed in sit-

uations where the static type of an object is not specific enough. (Type casts are not "safe" in the sense that they cause no type errors. They are "safe" in in the sense that any errors will be detected in a well-defined way, rather than causing upredictable results.)

In DEEP, we would like to go one step further, and use dynamic type checks to improve static typing. Consider the following Java-like pseudo-code:

```
void print(object o) {
  if (o instanceof String) {          // case 1
    System.out.println(o);
  }
  else if (o instanceof Integer) {    // case 2
    System.out.println(Integer.toHexString(o));
  }
  else System.out.println("<unknown object>");
}
```

In this example, the static type of o is object. However, after performing a dynamic type check using the instanceof operator, our knowledge of the type of o changes. Within the the case 1 code block, we can safely assume that o has type String, while within the case 2 code block, we can assume that it has type Integer. The Java language makes no such assumptions, so dynamic type casts would be required for the above code to be legal. However, type casts are not necessary if the static type system understands dynamic checks.

Because DEEP supports dependent types, ordinary equality comparisons may also introduce important type information:

```
zmap(T ≤ Top, s ≤ Int, f ≤ T → T → T,
     a1 ≤ Array(T,s), a2 ≤ Array(T,s)): Array(T,s) = ...;

n: Int = _;
m: Int = _;
a: Array(Int, n) = _;
b: Array(Int, m) = _;
... if (m == n) then zmap(Int, n, (+), a, b) else ...
```

In this example, the zmap function performs an element-wise binary operation over two arrays, and it requires that both arrays have the same size. The a and b arrays are not necessarily the same size, but it is still safe to add them together, so long a dynamic

check verifies that m == n before proceeding.

4.9.2.1 Type unification

Using dynamic type checks to improve static typing is difficult because it introduces multiple type constraints on a given expression. For example, in the Java code above, there would be two constraints on o: we have $o \leq$ object from the original declaration, and $o \leq$ String from the dynamic check. These two constraints need to be reconciled in some way. In the Java example, reconciliation is easy because String is a subtype of object, so the two constraints can be reduced to a single one: $o \leq$ String.

In more complex cases, placing an additional constraint on one variable may cause other variables to be constrained as well. For example, assume that $x \leq \mathsf{List}(T)$ and $x \leq \mathsf{List}(Int)$. If List is invariant in its argument type, then we know that $T \equiv$ Int. The propagation of constraints is somewhat similar to the unification algorithms used in type inference [Pierce, 2002], but there are three significant complications.

The first complication is that because DEEP supports subtyping, unification must deal with inequalities as well as equalities, which leads to more complicated relationships between variables.

The second complication is that in the presence of multiple inheritance, it may not be possible to reconcile certain constraints. For example, assume that $x \leq A$ and $x \leq B$, where A and B are unrelated. The constraints cannot be combined because they are unrelated terms, but they are not inconsistant either, since x may be a module that inherits from both A and B. The most obvious way to deal with the situation would be to use intersection types, e.g. $x \leq A\&B$, but intersection types are not a trivial addition to the type system.

The third complication is that constraints may be placed on expressions as well as variables, e.g.

```
if (foo(a) instanceof Integer) then foo(a)+1 else 0;
```

Putting constraints on expressions makes it very hard to develop efficient subtyping algorithms. Even more importantly, the safety of the technique is questionable. Any attempt to apply subtype reductions to arbitrary expressions, e.g. foo(a) $\xrightarrow{\leq}$ Integer, will likely lead to a loss of confluence, and perhaps allow a counter-example to transitivity elimination to be constructed.

On the other hand, restricting comparisons to variables would break the substitution lemma, so type safety in the presence of unification might be quite delicate.

4.9.2.2 Cast functions

One alternative to unification is to use explicit type casts. Under this scenario, the test (x instanceof t) would not merely return true or false. Instead, it would return a *cast function*, that, for any monotone function F, would cast an object of type $F(x)$ to an object of type $F(t)$. As discussed previously, such cast functions have been used to implement GADTs [McBride and McKinna, 2004].

By definition, type casts do not actually do anything. Every cast function is equivalent to the identity function, and can be safely erased from compiled code. In order to ensure such erasure, however, the core calculus must identify casts, and provide operations to safely construct and manipulate casts [Sulzmann et al., 2007].

The advantage of using type casts over unification is that no extensions to the type system are necessary. There is no need for intersection types, and no concerns about type safety. On the other hand, inserting type casts by hand is so tedious as to be unusable in practice, so some compiler support is clearly necessary.

A possible compromise would be to give the semantics of dynamic type checks in terms of casts, but insert the casts automatically by using a unification-like algorithm to figure out where casts are needed.

4.9.2.3 Practical application 1: pattern matching

The way in which type constraints propagate in a dynamic type check is identical to the way in which constraints propagate in pattern matching for GADTs, as discussed in Section 4.8.11.

Every pattern in a case statement can be easily represented as a type. If x is a list, and x matches the pattern Cons(h,t), then x must be a subtype of Cons(h,t). Some other languages, most notably Scala, also exploit this parallel; Scala allows types to be used as patterns in pattern matching expressions [Odersky and et. al., 2004].

Proper support for dynamic type checks would allow DEEP to encode GADTs, and thus solve the DSL expression problem in a functional programming style.

4.9.2.4 Practical application 2: Partial evaluation

Dynamic equality checks would also strengthen the partial evaluator. For example:

```
if (n == 2) then pow(x,n) else ...
```

The call to pow(x,n) can be inlined and partially evaluated if the value of n is statically known. In the current definition of DEEP, n would not necessarily be known in the above example. However, by using dynamic type checks, we can obtain a value for n from the test n == 2.

4.9.2.5 Dynamic type checks: summary of future work

In a static type system, subtype comparisons are ordinarily done statically (i.e. at compile-time). We would like to extend DEEP calculus so that such comparisons can be done dynamically (i.e. at run-time) as well. This extension is important because the result of a dynamic comparison can be used to strengthen static type judgements, thus allowing certain operations, like pattern matching over GADTs, to be statically type-checked. Although performing a comparison at run-time is relatively simple, extending the static type system is not, and will require more research.

4.9.3 The current implementation of DEEP

The last major area for future work that we wish to discuss does not concern the formal calculus or the theory of DEEP, but instead concerns its implementation. The mere existance of a calculus does not imply that the calculus can be implemented effectively in practice. The λ-calculus is prime example of how tricky implementations can be; although it is mathematically very simple, that simplicity is deceptive. Efficient implementations of functional languages rely on a variety of techniques, such as λ-lifting [Johnsson, 1985], thunks for lazy evaluation [Marlow and Peyton Jones, 1998], and closures to handle currying [Peyton Jones, 1992].

4.9.3.1 Status of the current implementation

We have implemented the DEEP calculus in order to demonstrate empirically that DEEP is more than just a theory, and can be used for practical programming. The current implementation is written in Scala [Odersky and et. al., 2004], and is about 5000 lines of code. We chose Scala because it supports pattern matching, but still compiles to the Java virtual machine and can thus make use of Java libraries.

The current implementation of DEEP is a proof-of-concept; it is nowhere near mature enough to be used for real-world projects. DEEP is implemented as an interpreter using standard term rewriting techniques. Our use of term rewriting means that the

interpreter is close to the core calculus, and was relatively easy to develop, but it is still very slow, even by the standards of interpreted languages.

Our objective in writing a proof-of-concept implementation was to tackle the issues that are most likely to be problematic. The current implemention of DEEP successfully implements the following:

Practical type-checking algorithms. The definition of well-formedness is syntax directed, and is thus fairly easy to implement, but the definition of declarative subtyping is not. We used the technques discussed in Section 2.8 to implement a subtyping algorithm. As discussed previously, there are actually two separate algorithms: $t \overset{\leq}{\longrightarrow\!\!\!\!\!\rightarrow}_{\min} v$ uses subtype reduction to determine the principle value type of t, while $t \leq_{\text{cnl}} u$ implements canonical subtype comparisons.

μ-lifting. We have implemented the μ-lifting algorithm described in Section 3.5.3. Subtype comparisons between modules are nominal rather than structural.

Lazy type checking. We have implemented the lazy type checking algorithm discussed in Section 3.6.2. Lazy type checking is necessary in order to handle recursive modules.

Linking. We implemented and tested the DEEP linking mechanism described in Section 4.5.7.2. (There is a significant caveat, though, which is described below.)

Partial evaluation. We have implemented the partial evaluation mechanisms discussed in Section 4.7, including the introduction and shuffling of let-expressions, λ-lifting, and forced specialization. The partial evaluator is a key component that interacts with the rest of the type system. The subtyping judgement uses the partial evaluator to reduce terms to normal form, and the linker uses it to reduce terms to well-linked forms. Partial evaluation must also be lazy, in order to implement lazy type-checking.

Although subtyping is the most complex part of the formal calculus, it is the simplest part of the implementation; the subtype relation comprises less than 300 lines out of the total 5000. The well-formedness judgement is likewise straightforward, although lazy type-checking is a significant complication. Since Scala is not lazy, we had to implement our own thunks. The μ-lifting operation is trivial, and is done during the initial parse phase.

The only portions of DEEP that were particularly difficult to implement are partial

evaluation and linking. The majority of the current code base is concerned with partial evaluation. Introducing, manipulating, copying, and garbage-collecting let-expressions turned out to be a major hassle, which took a great deal of work to resolve.

The linking mechanism is also quite tricky, because it involves tentatively performing reductions, and then discarding the result. We wrote and tested an initial implementation which validates the basic concept. However, that implementation was broken by subsequent changes to the way we handled μ-lifting and closures. As a result, link-checking is no longer implemented in the current code base. As discussed in Section 4.9.1, we are investigating better ways to do it.

As with any piece of code under development, the current implementation has a number of other known bugs and issues as well. For example, DEEP type-checking and partial evaluation are currently done simultaneously in a single pass. This means that some illegal cycles are not properly detected, polarity annotations are used for subtyping but not properly verified by well-formedness, and type error messages are often exceedingly long, since they involve partially-evaluated expressions.

As a result of these issues, we have not made the DEEP implementation available to the public. The implementation is robust enought to successfully type-check and partially evaluate our solution to the DSL-expression problem. However, it is not robust enough to withstand hacking and experimentation by other researchers. However, we would be happy to make the source code available on an as-is basis to interested parties who are aware of its alpha-software status.

4.9.3.2 Future improvements

There are a large number of ways in which the implementation of DEEP could be improved. Bug fixes and a re-implementation of the linking code are obvious necessities. In order to make polymorphic code usable in practice, we need to add support for automatically inferring implicit arguments (see Section 4.8.9). A Haskell-style mechanism for inferring type class arguments would be also be useful, although it is less critical. Syntax sugar for variant data types and pattern matching would be welcome improvements as well, as would support for monadic IO.

In the longer term, we would like to implement a compiler which translates DEEP to one of several available virtual machines, such as the JVM, .Net, or LLVM [Lattner, 2002]. Such a virtual machine would give access to Java, .Net, or C software libraries from within DEEP. Virtual machines like .Net and LLVM also support run-time code generation, which could dramatically speed up the partial evaluator.

4.10 Conclusion

The DEEP calculus extends DEEP-- by adding support for mixins and virtual inheritance. It does so by making two changes:

1. Functions and slots can be marked with a polarity, in order to specify subtyping relationships more precisely.
2. The parent of a module no longer needs to be statically known. It can be any term, including variables or virtual classes.

4.10.1 Polarity and multiple inheritance

The addition of polarity to the subtype relation is a relatively simple extension. The basic concepts have already been developed for System $F^{\omega}_{\leq}$ [Steffen, 1997] and are well-understood. Polarity adds very little complexity to either the theory or to the meta-theory of DEEP.

Our main contribution in this area is merely an observation. We have shown that polarized higher-order subtyping provides an elegant way of encoding multiple inheritance, an issue that has confounded object-oriented languages since their inception. The problems with multiple inheritance are currently regarded as so severe that more recent mainstream OO languages, such as Java and C#, have abandoned multiple inheritance altogether in favor of interfaces. We believe that language designers were too quick to abandon multiple inheritance. There is a practical need for the ability to combine classes together at the implementation level, a need that interfaces do not satisfy.

Mixin-style inheritance allows class implementations to be combined in a convenient manner, but previous attempts to implement mixins suffered from several drawbacks. First, most languages which support mixins, such as CLOS and Dylan, rely on automatic linearization, and that makes it difficult to control or even predict the order in which mixins are applied [Barrett et al., 1996]. The ordering problem can be resolved by defining mixins explicitly as functions over classes, as can be done with C++ templates [Smaragdakis and Batory, 2002], or as dedicated language extensions [Ancona et al., 2000]. However, defining mixins as functions creates a typing problem. What is the type of a mixin, and how should subtyping over mixin compositions be defined? Ancona, Lagorio and Zucca identify this as one of the biggest outstanding problems with their addition of mixins to Java [Ancona et al., 2000].

Polarized higher-order subtyping solves this problem in an elegant way. A mixin is any function which is a subtype of the monotonic identity function on classes, written $\lambda^{+} x \leq t.\ x$. Using this definition, we can show that all of the usual subtyping rules for multiple inheritance still hold, except that the rules take ordering and duplicates into account.

4.10.2 Linking

Virtual inheritance, which is the second extension introduced in DEEP, turns out to be much more complicated than polarity. On the surface, the extension looks innocent enough. After all, it does not involve any immediate changes to the syntax of DEEP-- at all, it merely eliminates a troublesome restriction. Nevertheless, eliminating the restriction introduces major complications to both the theory and the meta-theory of DEEP.

The fundamental problem with virtual inheritance is the fact that a composition can introduce name clashes, and name clashes are very difficult to detect statically. The easiest solution would be to detect them dynamically instead, but doing so would violate type safety, because the presence of name clashes can make certain subtyping judgements invalid (see Section 4.5.6).

We have introduced a linking model which statically detects name clashes. Unfortunately, linking in DEEP is somewhat unintuitive, and much more complex than we would like it to be. This complexity is largely hidden from the programmer, because the linking annotations can be automatically inferred, but as type theorists we are still searching for a more elegant model.

4.10.3 Applications of DEEP

The addition of mixins and virtual inheritance to DEEP allows us to solve the DSL expression problem. We would like to emphasize that the DEEP calculus is not a hypothetical language, nor is our solution to the DSL expression problem a hypothetical solution. We have implemented the calculus, the type theory, and the partial evaluator, and we have tested our solution to the DSL expression problem with the current implementation.

A working implementation does not demonstrate that the type theory of DEEP is sound. Nor have we run experiments that are large enough to demonstrate that that DEEP is suitable for real-world software engineering. However, we have demonstrated

that the DEEP calculus and its associated type theory are usable in practice, and that it is capable of solving problems that, to our knowledge, cannot be solved by any other language.

Chapter 5

Related Work

In theory there is no difference between theory and practice, whereas in practice, there is.

— Source unknown. This quote has been variously attributed to Jan L. A. van de Snepscheut, Yogi Berra, Chuck Reid, and Albert Einstein.

5.1 Introduction

This chapter provides a brief survey of previous work in the literature that is related to either pure subtype systems or DEEP. The design of DEEP draws inspiration from a wide variety of sources which are not confined to a single discipline. One of the biggest challenges in our work has been our desire to tread the line between theory and practice, hence the above quote. Since our work touches on so many areas, a complete survey of the literature is not feasible. Instead, our focus in this chapter will be on how DEEP and pure subtype systems fit into the bigger picture, and how our work differs from what others have done before.

The concept of "pure subtype systems", and our presentation of System $\lambda_{\lhd}$ in particular, are built on a large body of theoretical work in logic, type theory, and the formal semantics of programming languages. However, the design of the DEEP-- and DEEP calculi is also inspired by object-oriented programming.

Object-oriented languages are widely used, and a great deal of practical work has been done on the design and construction of various languages and systems. However, theoretical models of OO languages are generally more difficult to construct than models of functional languages. In part this is due to the fact that most OO languages

are large, complex, and designed for practical deployment, without much regard for theoretical concerns. However, OO languages also make extensive use of mechanisms that are theoretically challenging, such as subtyping, inheritance, and mutable state.

In addition to object-oriented programming, our work is also related to a several other software engineering techniques. A large body of work has been done on partial evaluation, generative programming, and domain-specific languages. In recent years, a whole research community has sprung up to study aspect-oriented programming and feature-oriented programming, which are also generative programming techniques.

Some of the literature that is most directly related to our work has been mentioned in previous chapters, and we will not go over it again here. Instead, we will focus on some highlights in the literature that we have either not discussed yet, or have mentioned only in passing.

5.2 Type Theory

Type theory is a large and active discipline in which new work is constantly being done. Types are interesting, in part, because they bridge the gap between mathematical logic and practical programming. The humble type-checker, found in compilers everywhere, is a tool for automated reasoning about programs. In fact, it may be the most successful real-world application of automated reasoning in the field of computer science. Types are familiar to most programmers, and can be used to analyze programs that range in size from a single line of code to millions of lines of code.

Researchers have been quick to capitalize on this success, and now routinely use types to anyalyze a variety of program properties that go well beyond simple datatypes like Int and String. Examples include ownership, memory and resource use, security, side effects, and others. However, such uses of types are well beyond the scope of this thesis, and will not be mentioned further.

5.2.1 The type/object dichotomy

Much of type theory in the literature is built on a dichotomy between types and and objects. The previous chapters have discussed how types and objects can be unified, and given some examples of why we might wish to do so. Nevertheless, there are both theoretical and practical reasons for enforcing a strict dichotomy; we discuss those reasons here.

5.2.1.1 Types as sets

On the theory side, there are number of parallels between type theory and logic and mathematics. Types are often interpreted as sets — such as the set of natural numbers, or the set of all strings — and objects are elements of a set. This interpretation scales well to higher-order type theories: kinds are sets of types. The "types as sets" model draws a parallel between the functions in a programming language, and functions in set theory; a function of type $A \rightarrow B$ is a mapping between the elements of set A and set B.

The parallel with set theory is not exact, however, because types are not nearly so versatile as sets. Most type systems do not allow concepts like the "set of prime numbers" to be represented as a type. Given a function f of type $A \rightarrow B$, the range of f is usually a much smaller set than the set of all objects of type B. In practical programming the domain of f may also be smaller than A; integer division, for example, requires the divisor to be a "non-zero integer" — a concept that cannot be easily represented as a type.

Types are more limited than sets because they are syntactically defined; typing is a syntactic relation between terms. As a result, it is possible to consider other syntactic relations (like subtyping) as a possible replacement for typing.

5.2.1.2 Types as propositions

Another theoretical reason for the dichotomy is the Curry-Howard isomorphism [de Groote (editor), 1995], which states that types correspond to logical propositions, while objects correspond to proofs of those propositions. The parallel between types and propositions is much deeper than the parallel between types and sets, because propositions are syntactic.

According to the Curry-Howard isomorphism, every type system corresponds to a logic. The concepts which can be represented as types within a given type theory are exactly the same as the concepts which can be written down as propositions with the corresponding logic. The reasoning power of the type system is the same as that of the corresponding logic.

As was discussed in Section 2.4.4, System $\lambda_{\triangleleft}$ is not strongly normalizing. We proved this fact by embedding System $\lambda*$, a type theory that is logically inconsistant according to the Curry-Howard isomorphism. The presence of Top in System $\lambda_{\triangleleft}$ makes the theory fully impredicative. Although we have not proved it, we believe that

strong normalization could be restored to System $\lambda_{\lhd}$ by removing Top, which would make the theory fully predicative.

Unfortunately, because there is no typing relation in System $\lambda_{\lhd}$, it is not yet clear how the Curry-Howard isomorphism could be applied, even in a predicative theory. We expect that pure subtype systems correspond to a logic of some kind, but it is hard to say what kind of logic that would be.

We further note that although the Curry-Howard isomorphism is an important concept, a strict interpretation of types as propositions is not necessarily accepted even within traditional type theory. In his development of the extended calculus of constructions, Luo writes:

> It is our view that *it is not natural to identify types with propositions*. It seems to the author that there are apparent differences, both conceptual and structural, between the purely logical entities and the others such as computational and mathematical, although it is sometimes admittedly difficult to draw a clear line between them. ([Luo, 1994], pp. 13)

Luo argues that although it is possible to use propositions as data types, such an encoding has certain weaknesses. For example, in the standard encoding of Church numerals, there is no linear-time predecessor function, and certain properties of the natural numbers cannot be proved within the axioms of the system. Luo thus provides additional facilities for defining inductive data types that are not propositions (and thus do not rely on impredicativity).

The relationship between types and objects, propositions and proofs, subtyping, and impredicativity is quite delicate. Understanding this relationship more fully is clearly an area for future work.

5.2.1.3 The phase distinction and type erasure

A practical reason for distinguishing between types and objects is that the dichotomy gives rise to a *phase distinction* [Harper et al., 1989]. The phase distinction states that type-checking and program execution occur in two distinct phases, which roughly correspond to compile-time and run-time. Type checking (i.e. the typing relation) is done at compile-time, while program evalution (i.e. reduction) is done at run-time.

Statically-typed languages are those which obey the phase distinction, and do all type-checking at compile-time. Type checking is an expensive operation, so doing checks ahead of time can yield a large increase in program efficiency. In addition, many statically typed languages gain a further advantage by performing *type erasure*.

Type erasure is a process that strips all of the type annotations out of a program [Pierce, 2002]. Even if type checks are not performed, type information has a cost in terms of both memory and execution time. The memory cost comes from tagging run-time values with their type. The run-time cost comes from that fact that in a polymorphic type system, types must be passed as arguments to polymorphic code. Type erasure eliminates the need to store tags or pass type arguments.

System $\lambda_{\triangleleft}$, DEEP--, and DEEP are all statically typed. Type-checking is done at compile-time, using the well-formedness and subtype relations. However, because we do not distinguish between types and objects, type erasure would be very difficult; there is no easy way to determine which terms are used as types and can be safely eliminated. This is a practical issue that may limit the ability of a DEEP compiler to optimize code.

5.2.1.4 Blurring the phase distinction

Having a strong phase distinction can be a useful asset in certain situations. Any computations at the type level are guaranteed to happen at compile-time, since that is when type-checking is performed. Type erasure also provides a strong guarantee to the programmer that a type-level computation is, in fact, fully static. In the presence of type erasure, types cannot affect run-time execution in any way.

Some languages exploit this property to good effect. For example, although originally intended as a simple mechanism for polymorphism, C++ templates have been widely used for sophisticated meta-programming and code generation [Veldhuizen, 1999]. Tim Sheard's Ωmega language uses the phase distinction somewhat differently. Although it is not dependently typed, types in Ωmega have a similar expressive power. Type erasure guarantees that the extra precision offered by the Ωmega type system does not incur any run-time penalty [Sheard, 1994] [Sheard, 2005].

On the other hand, there are a variety of language features that deliberately break the phase distinction in various ways. By breaking the phase distinction, a language can perform *phase shifting*: computations can be shifted from run-time to compile-time, or vice versa. Some examples are:

In Java, instances of class types retain type tags at run-time [Gosling et al., 2005], and these tags have a number of practical uses. Java combines the benefits of static and dynamic typing by allowing unsafe type-casts, but checking them at run-time. In essence, such type casts allow allow type-checking to be shifted from compile-time to run-time. It is also possible to branch on the run-time type of an object using the

instanceof keyword, and the getClass() method will retrieve the class of an object in a form that can be further queried using the Java reflection mechanism.

In a dependent type system, comparing types may involve evaluating object expressions, which shifts computations from run-time to compile-time. If types are indexed by objects, then it is also more difficult to perform type erasure. Some objects are only used for the purpose of static typing. Type erasure should clearly eliminate such objects, but it is difficult to distinguish between "objects for typing" and "objects for computation". An erasure algorithm that works in certain cases is presented in [Brady et al., 2003], but this algorithm does not work in all cases.

Partial evaluation and other generative programming techniques shift computations from run-time to compile-time for the purpose of efficiency [Jones et al., 1993]. Multi-stage languages with run-time code generation, such as Meta-OCaml [Taha, 2003], are even more flexible. Meta-OCaml provides support for multiple stages; there are explicit constructs for either deferring computations to a future stage, or shifting computations from a future stage to the current stage.

The DEEP calculus was designed to support both dependent types and partial evaluation. As such, it is not surprising that the phase distinction in DEEP is somewhat blurred.

5.2.2 Dependent types

Dependent types arise naturally as one side of Barendregt's λ-cube [Barendregt, 1992]. The λ-cube categorizes various typed λ-calculi according to which abstractions they support. Abstractions can be:

1. Functions from objects to objects (all calculi).
2. Functions from types to objects (polymorphic calculi – e.g. System F).
3. Functions from types to types (higher-order type systems – e.g. System F_ω).
4. Functions from objects to types (dependent types – e.g. the calculus of constructions).

The first form of abstraction corresponds to ordinary functions, and is supported by all calculi. The remaining 3 abstractions form the three axes of the λ-cube. It is possible to have each form of abstraction independently, so there are a total of 8 different combinations. Coquand's Calculus of Constructions [Coquand and Huet, 1988] supports all four forms of abstraction, and is the most sophisticated calculus on the

λ-cube.

5.2.2.1 Impredicativity

The calculus of constructions is an impredicative type theory. It differs in this respect from Martin Löf's intuitionistic type theory [Martin-Löf, 1984], which supports an infinite hierarchy of predicative universes (objects, types, kinds, types of kinds, etc.) in an effort to prevent the potential paradoxes that impredicativity can create. A π-type in Martin Löf's theory can only abstract over smaller types than itself. Predicativity means that dependent types are allowed, because the universe of objects is smaller universe than the universe of types, but polymorphism is restricted, because an object cannot abstract over types larger than itself.

Luo's Extended Calculus of Constructions (ECC) [Luo, 1994] combines both approaches within a single theory: it includes a single impredicative universe of propositions, with a hierarchy of predicative universes of types built on type of it. The impredicative structure of ECC is quite delicate. For example, although ECC supports strong existential sum types, such types are not propositions, and cannot be impredicative; it is known that impredicative existentials also lead to paradox [Hook and Howe, 1986].

5.2.2.2 Inductive Data Types

Luo's UTT type theory [Luo, 1994] further extends ECC with *inductive data types*, such as lists and natural numbers. Although such data types can be encoded directly in ECC using Church encodings, the Church encodings are somewhat weak. It is not possible to prove certain basic properties of Church encodings within the type theory, and the universe of propositions cannot be easily extended without inviting paradox. Thus, UTT adds inductive data types to the predicative type universes, much like Martin Löf's type theory.

Inductive data types are similar to ordinary data types in Haskell, ML, and other functional languages [Hudak et al., 1999], with two notable differences. First, since UTT does not support general recursion, inductive data types are the types of finite trees; it is not possible to represent infinite streams or circular data structures. Second, inductive data types are not necessarily uniform.

In Haskell 98, a recursive data type may only refer to itself using the same parameters with which it was declared. An inductive data type, on the other hand, may refer to

itself with different parameters. For example, the following Haskell-like syntax defines the type of vectors of type a and length n:

```
data Vector a (n::Nat) where
  VNil  :: Vector a 0
  VCons :: a -> (Vector a (m+1)) -> (Vector a m)
```

Notice that the two constructors VNil and VCons return different vector types. Vectors are uniform with regard to the type parameter a; a vector of type a is made up of subvectors of type a. However, vectors are not uniform with regard to the parameter n; a vector of length n contains a subvector of length n-1.

This non-uniformity makes inductive data types very expressive. However, it also makes pattern matching over inductive data types much more complicated. The result of a pattern match will change the types of various arguments. For example:

```
map :: (a -> b) -> (Vector a n) -> (Vector b n)
map f VNil         = VNil                    -- must show that n = 0
map f (VCons x xs) = VCons (f x) (map f xs) -- must show that n = m+1
```

In this example, the result of the pattern match will tell us the value of n. If the second argument is VNil, then $n = 0$, otherwise $n = m+1$ for some m. This information cannot be simply ignored; it is required in order to prove that the above example is well-typed. In the first case, VNil has type (Vector b 0), but it is supposed to have type (Vector b n), so it is well-typed only if $n = 0$.

5.2.2.3 Witnesses

In most type theories, pattern matching is not a core construct of the theory, but is instead implemented by means of an encoding. The standard technique is to exploit the Curry-Howard isomorphism; given two terms t and u, the proof that $t = u$ is a function that casts any term of type t to one of type u. The proof object is passed as a hidden argument, and casts are inserted to adjust types appropriately [McBride and McKinna, 2004].

A large number of interesting properties can be handled in this way beyond simple type equality. Any property of interest is declared as a type, and the proof of that property, often called a *witness* to the property, is an object of that type. For natural numbers, it is possible to construct proofs that addition is associative and commutative, that one number is less than another, and so on [McBride and McKinna, 2004].

Although the use of witnesses is a powerful idea, it is sound only for languages that are consistent, and thus strongly normalizing. The use of witnesses in languages like

Cayenne [Augustsson, 1998] and Ωmega [Sheard, 1994], which allow general recursion, is somewhat suspect, because a witness to any property can be easily constructed as a non-terminating expression. Ωmega claims to resolve this issue by using a strict evaluation rather than lazy evaluation, but the soundness of this approach has not yet been proven [Sheard, 2005].

5.2.2.4 Practical uses of dependent types

Dependent types are widely used in formal logic and theorem proving, and most research on dependent types has been conducted within this area. Dependent types form the basis of the Edinburgh logical framework [Harper et al., 1993] and Twelf [Pfenning and Schürmann, 1999], as well as the Coq and Lego proof assistants [Team, 2006] [Luo and Pollack, 1992], among others.

Dependent types are far less widely used in general-purpose programming languages, although there have been several notable attempts in recent years. The basic problem is that if types depend on objects, then general recursion at the object level leads to undecidability at the type level. Several ways of approaching this problem have been proposed.

Cayenne is a dependently-typed extension of Haskell [Augustsson, 1998]. Like DEEP, Cayenne does not attempt to control general recursion, and typing is undecidable as a result. The use of Cayenne as a logic — i.e. the use of witnesses to various logical properties — is somewhat suspect.

Epigram is a dependently-typed language which takes the opposite approach [McBride and McKinna, 2004]. All functions in Epigram must be both total and terminating. This means that Epigram can be used as a logic, but it is not Turing complete. Future plans for Epigram include trying to capture non-termination as a computational effect within a monad (unpublished conversation).

Dependent ML is a dependently-typed version of ML [Xi, 1998]. The strategy used in ML is to restrict dependent types in a certain way. Types cannot be indexed by any arbitrary object expression; they can only be indexed by certain types of object, such as integers and natural numbers. Basic axioms about numbers are built-in to the type system. This approach has been quite successful at using dependent types to perform certain common optimizations, such as elimination of array bounds checks [Xi, 2003].

The type systems of C and C++ are actually dependently typed, although this feature is rarely advertised as such. Integer constants are used to specify the length of array types in C, and can be passed as template parameters in C++ [Ellis and Strous-

trup, 1990]. This makes C and C++ the most widely used dependent type system by far.

5.2.3 GADTs

Generalized Abstract Data Types (GADTs) are an idea that is conceptually related to dependent types [Peyton Jones et al., 2006]. The technique has been explored in the literature under a variety of names, including Guarded Recursive Datatype Constructors [Xi et al., 2003], and first-class phantom types [Cheney and Hinze, 2003].

GADTs lift the uniformity restriction on data types in Haskell 98 and ML. GADTs are not technically dependent types, because they cannot be parameterized by objects. However, GADTs do use non-uniform constructors, and they require the complex pattern matching mentioned earlier. GADTs are strong enough to implement a well-typed interpreter, and thus solve the tag elimination problem. However, they cannot be used to encode true dependent types, like vectors of length n.

The Ωmega language combines GADTs with infinite hierarchy of type universes, and allows inductive data types to be declared within any universe [Sheard, 2005]. Although it is not possible to define the type of vectors of length n in Ωmega, where n is an object with type Nat, it is possible to define vectors of length N, where N is a type with kind Nat', which largely amounts to the same thing.

A similar approach is often used in Haskell. Unlike Ωmega, Haskell does not support variant data types and pattern matching in higher universes. However, Haskell's type class mechanism is sufficiently powerful to perform arbitrary computations over types [Kiselyov et al., 2004].

Sheard argues that the combination of GADTs plus universes makes dependent types unnecessary in Ωmega [Sheard, 2005]. Any inductive data type can be used as a type parameter simply by lifting it into a higher universe. The advantage of this approach is that it allows type erasure to be easily performed. The disadvantage is that every data type, and every operation on data types, must be redefined in a higher universe before it can be used as a type parameter.

5.2.4 Singleton types

Singleton types are another idea that is conceptually related to dependent types [Stone, 2000]. In Stone's formulation, a singleton type is written $\mathbf{S}(t : T)$, and is the type of

all objects of type T that are exactly equal to t. Singleton kinds are defined similarly; $\mathbf{S}(T : K)$ denotes a type of kind K that is equal to T.

Stone uses singleton types and kinds in much the same way as we use final bindings in DEEP and DEEP--. Singleton types are a way of implementing transparency or translucency in modules. If m is a module with type $\{x : \mathsf{Int}; y : \mathbf{S}(3 : \mathsf{Int})\}$, then $m.y$ must be equal to 3. Transparent type definitions play an important role in practical type-checking for modules (see Section 3.4.2), while transparent object definitions allow cross-module inlining [Stone, 2000].

Just like dependent types, the presence of singleton types means that type equality depends on object equality; if the object language is recursive, then typing is undecidable. Singleton types can also mimic dependent types even more directly, since any object can be packaged up as a type and used as a type parameter.

5.2.5 λ-typed λ calculi

Although pure type systems use a uniform syntax for terms at both the type level and at the object level, they still distinguish syntactically between functions and proper types. Functions are written with λ, e.g. $\lambda x : t.\ u$, while the type of a function is written with Π, e.g. $\Pi x : t.\ u$. The interpretation of functions and types is also very different; a function is a computational entity that can be eliminated with β-reduction, while a Π-type is a constant.

As we noted in Chapter 1, there is an underlying symmetry between functions and types. Both λ and Π are binders which introduce a new variable. The subtyping rules for Π-types and λ-abstractions are essentially the same. Moreover, the subtyping rules are similar to the typing rule that assigns a Π-type to a λ-abstraction. System $\lambda_{\lhd}$ exploits this symmetry by unifying Π-types and λ-abstractions; the supertypes of a λ are also λs.

Although the unification of typing and subtyping that we propose in this thesis is new, the unification of λ and Π is not. Hints of unification date back to Automath family of languages proposed by De Bruijn in the 1960s and 70s [de Bruijn, 1970], where the syntax $[x : A]B$ is used to denote both $\lambda x : A.\ B$ and $\Pi x : A.\ B$.

Somewhat more recently, the λ^{λ} calculus also uses λ in place of the Π-binder [de Groote, 1993]. An ordinary pure type system such as the calculus of constructions has only three sorts of term: functions, types, and kinds, e.g. $(\lambda x : T.\ u) : (\Pi x : T.\ U) : *$. In contrast, the λ^{λ} calculus permits an arbitrary sequence of typing judgements: $(\lambda x :$

$T.u_1) : (\lambda x : T.u_2) ... : (\lambda x : T.u_n)$.

Fairouz Kamareddine and her colleagues have systematically studied the relationship between λ and Π in a series of papers. In [Kamareddine and Nederpelt, 1996], they present a λ-calculus which uses the following two rules:

$$\frac{\Gamma \vdash\ t : (\Pi x : a.\ b), \quad u : a}{\Gamma \vdash\ t(u) : (\Pi x : a.\ b)(u)} \qquad (\Pi x : a.\ b)(c) \longrightarrow [x \mapsto c]b$$

The first rule changes the way in which types are assigned to applications, while the second allows Π-types to be eliminated just like λ-abstractions. Kamareddine presents the following reasons for wishing to make λ and Π behave similarly:

1. Substitution. The standard formulation of pure type sytems has two forms of substitution: β-reduction, and the type rule for application. This seems redundant; a unified system would use β-reduction for all substitution.
2. Compatibility. In a unified system, the typing rule for application is *compatible* with the typing rule for functions — $f : F$ implies $f(a) : F(a)$, just as $u : U$ implies $(\lambda x : T.\ u) : (\Pi x : T.\ U)$.
3. Unified treatment of objects and types. Pure type systems already unify types and objects to a large degree, since functions over types and functions over objects have the same syntax. This unification can be taken further by making λ and Π behave similarly.
4. Distinguish between two important questions of typing: (A) Is a term t typeable? (B) What is the type of t?

Kamareddine's first three arguments are very similar to the symmetry arguments that we presented in Chapter 1. Even without subtyping, it is clear that there is a symmetry in pure type systems that has not been properly exploited; subtyping simply makes this symmetry more apparent. Kamareddine's fourth argument is also addressed in this thesis, because System $\lambda_{\lhd}$ formulates subtyping and well-formedness as separate judgements.

Kamareddine's fourth argument is repeated by Peyton Jones and Meijer in [Peyton Jones and Meijer, 1997] for practical, rather than theoretical reasons. In a compiler, the type-checking phase verifies that all terms are well-formed. However, the type of a term may be required in subsequent phases as well, even after it is known to be well-formed. Peyton Jones and Meijer thus use β-reduction on Π-types for the purpose of calculating types independently of type-checking.

The system presented in [Kamareddine and Nederpelt, 1996] suffers from a severe drawback. The type $(\Pi x : a.\ b)(u)$, which is assigned to an application, is reducible

but is not well-formed. Kamareddine resolves this issue first by using let-bindings in [Kamareddine et al., 1997], and then by completely eliminating the difference between λ and Π altogether in [Kamareddine, 2005]. She shows that the unification of λ and Π preserves the most important properties of all systems in the λ-cube, including type safety and strong normalization. The only property that is not preserved is unicity of types. The term $\lambda x : t.\ u$ (given that $u : *$) can either have type $\lambda x : t.\ *$, if it is acting as a function, or $*$, if it is acting as a Π-type.

In the embedding of pure type systems into System $\lambda_{\triangleleft}$ that we give in section 2.4, the lack of unicity of types is just an ordinary consequence of subtyping; it is reflected in the fact that $(\lambda x : t.\ u) \leq (\lambda x : t.\ \mathsf{Top}) \leq \mathsf{Top}$.

5.3 Subtyping

The relationship between System $\lambda_{\triangleleft}$ and several subtype systems is discussed at length in Section 2.9.2; we will not repeat it here. What follows is a brief overview of related work in the theory of subtyping that was not discussed previously.

Like typing, the idea of subtyping has parallels within logic and mathematics. If types are viewed as analogous to sets, then the subtype relation is analogous the subset relation. If types are viewed as propositions, then subtyping corresponds to implication. For example, the standard subtyping rule for arrow types is written as:

$$\frac{\Gamma \vdash\ B \leq C}{\Gamma \vdash\ A \to B \leq A \to C}$$

This rule can be interpreted logically as meaning that if B implies C, then a proof that A implies B can be used to construct a proof that A implies C.

5.3.1 Coercive subtyping

Formal definitions of subtyping can be divided into two major groups: *inclusive subtyping*, and *coercive subtyping*. In inclusive subtyping, if $B \leq C$, then a program of type B can be used in any position where a program of type C is expected. Inclusive subtyping is related to the interpretation of types as sets; an element of a subset is automatically considered to be an element of a superset. System $\mathrm{F}_{\leq}$, $\mathrm{F}^{\omega}_{\leq}$, most object-oriented languages, and the systems considered in this thesis all use inclusive subtyping.

Coercive subtyping is related to the interpretation of types as propositions [Luo, 1999]. If B implies C, then a proof of that fact is a function that constructs a proof of C when given a proof of B. As far as subtyping is concerned, this means that if $B \leq C$, then there must exist a function that casts an object of type B to an object of type C. A language without subtyping can mimic a language with subtyping by inserting type casts at any point where an object of one type must be coerced to a different type.

The advantage of coercive subtyping is that it is possible to define coercions between data types that would not otherwise be related. For example, many general-purpose languages, such as C and Java, automatically insert coercions between certain built-in data types, such as casts from int to double. However, such coercions are only safe if the entire meaning of the object is preserved. For example, it is a an error to automatically cast from int to float, because an int has 32 significant bits, while a IEEE float has 23.

Due to considerations such as these, the meta-theory of coercive subtyping is actually quite similar to that of inclusive subtyping. The most important meta-theoretical property for a coercive system is *coherence*, which states that if $B \leq C$, then the coercion from B to C must be unique [Luo and Luo, 2001]. If the coercion was not unique, then there would be several possible coercions that could potentially be applied, with no way to choose between them.

Uniqueness can be ensured by formulating the subtyping judgement in an algorithmic, or syntax-directed manner. If the subtyping judgement is syntax-directed, then every subtype derivation must be unique. A unique coercion can then be generated from the derivation. The transitivity rule is not syntax directed, so transitivity elimination is just as important for coercive subtyping as it is for inclusive subtyping.

In System $\lambda_{\lhd}$, subtype derivations need not be unique. However, we have formulated the subtype relation as an abstract reduction system. If the underlying reductions commute, then the exact order of reductions doesn't matter; all reduction paths which connect the same two terms are guaranteed to be equivalent.

5.3.2 Circular dependencies in subtyping

In the simply typed λ-calculus and System $F_{\leq}$, all types which are syntactically legal are valid types. In more complex type systems, such as System $F^{\omega}_{\leq}$ and dependently-typed calculi, type expressions are not necessarily valid, and a kinding judgement is used to ensure that types are well-kinded. The meta-theory of subtyping in such sys-

tems is often much more complex, because there is a circular dependency between subtyping, typing, and kinding that is very difficult to unravel.

The change in complexity can be clearly seen in System $F^{\omega}_{\leq}$. The first formulation of System $F^{\omega}_{\leq}$, due to Steffen and Pierce, does not have bounded quantification on type operators [Steffen and Pierce, 1994]. The syntax of the system is as follows — note that the kinds K are quite simple.

$$
\begin{aligned}
t &::= x \mid \lambda x : T.\, t \mid \lambda X \leq .\, T.\, t \mid t(t) \mid t(T) \\
T &::= X \mid T \to T \mid \Pi X \leq T.\, T \mid \lambda X : K.\, T \mid T(T) \\
K &::= * \mid K \to K
\end{aligned}
$$

Typing depends upon subtyping in two ways, both because of the standard subsumption rule, and because polymorphic functions use bounded quantification, as shown below:

$$
\frac{\Gamma \vdash\ t : T \quad T \leq U}{\Gamma \vdash\ t : U}
\qquad
\frac{\Gamma \vdash\ t : \Pi X \leq U.\, S \quad T \leq U}{\Gamma \vdash\ t(T) : [X \mapsto T]S}
$$

Typing depends upon kinding to ensure that types are well-kinded. Subtyping also depends upon kinding for the same reason — $T \leq U$ only if T and U are well-kinded. Nevertheless, Steffen and Pierce's formulation manages to avoid a circular dependency because kinding does not depend upon anything. The meta-theory of the system thus has a clear order. One first establishes certain properties guaranteed by kinding, such as strong normalization of types, then establishes properties of subtyping, and then properties of typing.

In Compagnoni and Goguen's formulation of System $F^{\omega}_{\leq}$, type operators use bounded quantification, rather than kinding [Compagnoni and Goguen, 2003]. The kinds in the system are thus defined differently:

$$
\begin{aligned}
T &::= \ldots \mid \lambda X \leq T.\, T \mid \ldots \\
K &::= * \mid \Pi X \leq T.\, K
\end{aligned}
$$

Kinding in this system depends upon subtyping, since $T(U) : K$ only if $T : \Pi X \leq S.\, K$ and $U \leq S$. There is thus a circular dependency: subtyping depends upon kinding, and kinding depends upon subtyping. The circular dependency on judgements leads to a circularity in the meta-theory, because judgements cannot be analyzed separately.

As was discussed in section 2.9.2, current proofs of transitivity elimination for System $F^{\omega}_{\leq}$ depend upon strong normalization of types. The proof of strong normalization depends upon kinding, but the safety of the kinding system depends upon transitivity

elimination. Compagnoni and Goguen unravel these dependencies using the technique of typed operational semantics, in which strong normalization is a premise of the kinding and typing rules.

Aspinall and Compagnoni encounter a similar circular dependency when combining subtyping with dependent types [Aspinall and Compagnoni, 1996]. Dependent types introduce a dependency between kinding and typing. Typing depends upon subtyping, subtyping depends upon kinding, and kinding depends on typing. Aspinall and Compagnoni resolve the circularity by (1) defining the algorithmic subtype relation on pre-types, rather than well-kinded types, (2) splitting β-reduction into two parts, so type reduction can be considered separately, and (3) performing proofs in a very specific order.

When adding subtyping to pure type systems, Zwanenburg also defines subtyping over pre-terms in order to avoid the dependency between subtyping and typing [Zwanenburg, 1999]. However, a major consequence of this decision is that Zwanenburg is unable to include bounded quantification on type operators, because such quantification would be unsound on preterms. Zwanenburg also removes Top for the same reason: Top is generally considered to be a supertype only of types with kind $*$, and kinding information is not available on pre-terms.

Relevance: Our definition of System $\lambda_{\lhd}$ has a circular dependency between subtyping and well-formedness. Like Aspinall and Zwanenburg, we define an algorithmic subtype relation over pre-terms in an effort to break the circularity. Our definition of algorithmic subtyping differs from Aspinall and Zwanenburg's in a few key respects. First, we add a premise to the β-reduction rule: $(\lambda x \leq a.\ b)(c) \xrightarrow{\equiv} [x \mapsto c]b$ only if $c \leq *a$. This premise allows us to safely use bounded quantification on type operators, even when the terms in question are not well-formed. We resolve the issue with Top by making Top range over all terms, not just those of kind $*$, but Girard's paradox is the cost of this change.

5.3.3 Singleton types

In addition to studying the combination of subtyping with dependent types, Aspinall has studied the combination of subtyping with singleton types in [Aspinall, 1994]. Instead of using Stone's syntax of $\mathbf{S}(t : T)$ for a singleton type, Aspinall writes $\{t\}_T$, but the meaning is similar.

From our point of view, the most interesting thing about subtyping with singletons

is that it highlights yet another symmetry between typing and subtyping: $\{t\}_T \leq T$ if and only if $t : T$. Aspinall explicitly notes that because of this symmetry, any typing judgement can be formulated as a subtype judgement, and vice versa.

Relevance: System $\lambda_{\lhd}$ has no special syntax for singleton types. However, the idea of singletons features prominently in our work, because every term can be interpreted as either a type or an object depending on context. When an object like the number 3 is interpreted as a type, it behaves like a singleton type. It is not hard to see that the standard Church-encoding of 3 has only trivial subtypes; see Section 2.3.3.

5.3.4 Power types

Much of the theory of higher-order subtyping, and subtyping with dependent types, was inspired by Cardelli's early work on *power types*. The approach take by Cardelli with power types is exactly opposite to the one that we have pursued with pure subtype systems. Rather than treating typing as a special case of subtyping, power types allow subtyping to be treated as a special case of typing. Nevertheless, the overall effect is very similar to pure subtype systems, both in terms of expressive power, and in complexity (i.e. intractibility) of the meta-theory.

In Cardelli's system, the power type of T, written $\mathit{Power}(T)$, is the kind of all subtypes of T. Power types thus support both typed quantification and bounded quantification with a single syntax. The function $\lambda x \leq t.\ u$ can be written as $\lambda x : \mathit{Power}(t).\ u$.

Just like System $\lambda_{\lhd}$, Cardelli's system suffers from Girard's paradox, because it takes Type:Type as an axiom. Cardelli also adds full fixpoints to both the language of types and the language of objects, much as we do in DEEP-- and DEEP. The typing is undecidable, and Cardelli does not attempt to prove meta-theoretic results such as type safety.

More recently, Aspinall has reformulated power types as a predicative system which removes both fixpoints and Type:Type [Aspinall, 2000]. This reformulation is somewhat easier to analyze than Cardelli's original version, but even so, Aspinall was unable to prove type safety, for a reason that is very similar to the one we discuss in Chapter 2.

According to Aspinall, the basic problem is that the proof of type safety requires a *generation principle*. Given a particular typing judgement, it is important to know how that judgement was derived. In a type theory without subtyping, it is usually not too hard to formulate the typing rules in a syntax-directed manner, which allows the exact

derivation to be determined from the syntax of terms in the judgement. However, this is not necessarily possible in a type theory with subtyping and bounded quantification:

> The sticking point is bounded operator abstraction, which makes it hard to prove substitution lemmas in the syntax-directed system before proving other properties which depend on substitution [Aspinall, 2000].

Our development of the meta-theory for System $\lambda_{\lhd}$ suffers from exactly the problem that Aspinall describes. In Section 2.5.2, the necessary generation principle for System $\lambda_{\lhd}$ is called *inversion of subtyping*, and is a consequence of transitivity elimination. Transitivity elimination in the algorithmic system depends upon confluence, and confluence requires a substitution lemma.

The substitution lemma, in turn, makes use of transitivity, thus introducing yet another circular dependency. Assume we have a transitivity-free derivation of $\Gamma, x \leq u \vdash_A x \xrightarrow{\leq} u \leq s$. If we substitute t for x within this judgement, where $t \leq u$, then we obtain $\Gamma \vdash_A t \leq u \leq [x \mapsto t]s$. Performing the substitution introduces transitivity into a transitivity-free derivation.

We resolve this problem in System $\lambda_{\lhd}$ by including transitivity within the definition of algorithmic subtyping, and then attempting to show that any subtype derivation which relies on transitivity can be converted to a transitivity-free form. Our approach currently lacks a sufficiently cunning inductive hypothesis. An inductive hypothesis is needed in order to demonstrate that a substitution will only create new instances of transitivity that are "smaller" than the instance that is currently being eliminated.

5.4 Modules, Mixins, and Linking

Historically, the development of module systems has tended to lag well behind the development of programming languages. The C language, which was used to develop the UNIX operating system (a very large piece of software), does not have a module system at all. Linking of C programs is done after compilation, essentially as a post-processing step, with very little help from the language or compiler.

C++ namespaces [Ellis and Stroustrup, 1990], Haskell modules [Hudak et al., 1992], and Java packages [Gosling et al., 2005] provide some support for modules, but the primary purpose of a module in these languages is to ensure that names are unique. A definition named "foo" in one module should not accidentally conflict with another definition named "foo" in a different module. Unique names are required because the

underlying linking model does not differ all that much from C. At run-time, all definitions in all modules are simply combined together into a single global namespace to form a running program. A consequence of the global namespace is that connections between modules are hard-wired. Modules in these languages are not intended to be used as a principle form of abstraction.

The module system of Standard ML differs sharply from the simplistic systems described above [MacQueen, 1984] [Tofte, 1996]. Modules in ML are one of the principle forms of abstraction in the language, serving much the same role in ML as type classes do in Haskell, or classes do in Java and C++. Modules can be parameterized by other modules, and can be instantiated in arbitrary ways. Perhaps most importantly, modules in ML can be *reused*; it is possible to instantiate a module multiple times with different parameters. The precise way in which modules are linked together is specified by the programmer in code, rather than fixed by the linker. Because of this heritage, much of the theoretical work on advanced module systems has been inspired by Standard ML.

There are three things that make modules theoretically challenging. First, a suitably rich module language should support computations over modules. Second, modules may contain both type definitions and object definitions. And third, there is often a need for definitions in a module to be mutually recursive. (Standard ML does not support recursive modules, but it is an oft-requested feature [Crary et al., 1999] [Dreyer et al., 2001] [Dreyer, 2007]).

5.4.1 Computations with modules

One of the distinguishing features of ML modules is that the module system is formulated as a separate language which is layered on top of core ML. In other words, although modules are an important form of abstraction, they are not first-class values in the core language; it is not possible to pass a module as an argument to a function, or store it in a reference cell [Harper and Stone, 2000].

The advantage of this separation is that the complexities of the module system do not disrupt core language mechanisms like type inference. The disadvantage is that many of the constructs of core ML must be duplicated at the module level. Modules have their own type system; the type of a module is called a *signature*. Functions over modules are called *functors*, and have their own semantics, which are distinct from the semantics for ordinary ML functions.

Work by Russo has reconciled this difference to some extent [Russo, 2000]. Russo's formulation allows modules to be packed into a core ML term, where it can be manipulated by the core language, and then unpacked into a module again. However, this mechanism does not truly unify the module language and core ML; it can best be regarded as a compromise between having a completely separate module language, and a completely integrated one.

5.4.2 Translucent modules and dependent types

Type definitions within a module are problematic because the type system must be able to propogate type information across module boundaries. If M is a module with a type member named T, then the expression $M.T$ denotes a type. As a result, static type checking requires that module-level computations be done at compile-time. At the same time, however, modules also contain object members, which means that module equality may depend on object equality, which is ordinarily computed at run-time. Because modules contain both type and object members, the ML module system has many of the characteristics of a dependent type system, and is often modelled as such [MacQueen, 1986]. Maintaining a phase distinction in this environment requires a good deal of work [Harper et al., 1989].

Moreover, there are times when the exact definition of $M.T$ must be known for type-checking purposes, and times when it should be hidden for abstraction purposes. Manifest types [Leroy, 1994], translucent sums [Lillibridge, 1996], and singleton kinds [Stone, 2000] have all been proposed as a way of declaring the exact identity of type members. The DEEP-- and DEEP calculi use final bindings for this purpose (see Section 3.4.2).

Translucency is theoretically challenging because the identity of type members must be declared in the module signature rather than the module itself, e.g.

```
signature STRINGDICT = sig
  type keytype = string
  type elemtype
  type dict
  val lookup : dict -> keytype -> elemtype
end
```

Any module which implements STRINGDICT can specify arbitrary types for elemtype and dict, but keytype must be equal to string. It is also desirable to have at least a little bit of subtyping, so that if DICT is a signature that allows an arbitrary keytype, then an instance of STRINGDICT is also an instance of DICT [Lillibridge, 1996].

5.4.3 Recursive Modules

One of the main limitations of modules in standard ML is that they cannot be recursive, so a good deal of work has been done on adding recursion. The essential features of ML-style recursive modules are outlined in [Crary et al., 1999], and further refinements of their ideas can be found in [Dreyer et al., 2001], [Dreyer, 2005], and [Dreyer, 2007]. The definition of a recursive module in these systems has two parts: the module itself, and a *recursively dependent signature* for the module:

module ::= rec (X: *signature*) *module-definitions*
signature ::= rec (X) *signature-definitions*

In both cases, X is a the self-variable of the module. The type of X is provided by the module signature. Because modules may contain type members, a module signature may also refer to X recursively. Note that a recursively dependent signature is not an ordinary recursive type. In both the module definition, and the signature definition, X is a module (i.e. an object) variable, not a signature (i.e. a type) variable. A signature is thus a dependent type, because it is a type parameterized by an object.

The first challenge posed by recursive modules is the fact that ML has a strong phase distinction. Allthough a module is defined syntactically using a single fixpoint, recursive types (types in core ML) must be handled differently than recursive objects. In particular, recursive types have a *contractiveness* requirement that prevents circular definitions like type $T = T$, while recursive objects have no such restriction.

Crary et. al. use *phase-splitting* to divide a module into two parts: a fixpoint over type definitions in the module, and a fixpoint over object definitions. This phase-splitting is only possible because the core type system of ML (excluding modules) is not dependently typed; objects depend upon types, but not vice-versa. The result of phase-splitting in [Crary et al., 1999] is a set of equi-recursive type equations, for which type-checking is not known to be decidable. Dreyer attempts to further constrain recursive type equations to a decidable form, but the algorithms for detecting and eliminating cycles are not trivial [Dreyer, 2007].

The second challenge posed by recursive modules is referred to by Dreyer as "the double vision problem" [Dreyer, 2007]. Because the definition of a module is separate from its signature, the signature may not give sufficient type information to check the body of the module, e.g.

```
signature S = rec (X) sig
  type t
  val succ: X.t -> X.t
end
```

```
structure M = rec (X: S) struct
  type t = int
  fun succ (y: X.t) : X.t = y+1   (* Error *)
end
```

The trouble here is that although the module M declares t to be int, this information is not exported in the signature S. Thus, X.t is treated as an unknown type, which does not necessarily support addition.

In [Crary et al., 1999], recursive dependent signatures are required to be fully transparent both for this reason, and because full transparency is required to verify that recursive types are contractive. However, full transparency inhibits abstraction because it prevents a module from hiding the implementation of certain type members. Dreyer describes an elaborate mechanism for lifting this requirement in [Dreyer, 2007].

The double vision problem illustrates a practical difficulty with using recursive ML modules: the signature of a module must be specified separately, and signatures are often quite verbose. Nakata and Garrigue have resolved this issue by writing a type inference algorithm which can infer the signatures of recursive modules [Nakata and Garrigue, 2006]. Their algorithm deals with recursion by being lazy.

Relevance: Phase-splitting is not possible in a language with recursive modules and dependent types. In DEEP, a single fixpoint is used for both type and object definitions. Our use of a single fixpoint is much simpler, but clearly undecidable; without phase-splitting, there is no way of imposing a contractiveness requirement on types.

The DEEP and DEEP-- calculi resolve the "double vision" problem by using the module itself as its own signature. The type of "self" within a module is thus fully transparent. However DEEP supports abstraction via subtyping; the supertypes of a module can hide implementation details. DEEP also uses lazy type checking to deal with recursion, much like the inference done by Nakata and Garrigue.

5.4.4 Recursive modules with late binding

The most critical feature that distinguishes ML-style modules from DEEP modules is the fact that ML-style modules use early-binding, whereas DEEP modules use late-binding. Late binding has been explored in formal calculi by Wells [Wells and Vestergaard, 2000], and by Ancona and Zucca [Ancona and Zucca, 2002].

Ancona and Zucca represent a module much like a function with multiple named inputs and outputs. An import list specifies services that a module requires from other

sources, while an export list specifies services that it provides. It is also possible to define local definitions that are neither imported nor exported. The basic operations on modules are (1) *sum*, which merges two modules together, (2) *freeze*, which links inputs to outputs, (3) *reduct*, which can add or remove inputs and outputs, and (4) *select*, which extracts the output of a module. The calculus supports late binding because the imports of a module have an unknown value when the module is declared. It is possible to mimic OO-style overriding by suitable use of the supplied operators [Ancona and Zucca, 2002].

In Wells' m-calculus, a module is a recursive record which has exported members, private members, and deferred members. Deferred members are names which are declared but not defined; they act much like the imports of Ancona and Zucca. The m-calculus differs from that of Ancona and Zucca because linking is performed automatically. When two modules are merged together, the provided members of one module will override deferred members in the other which have the same name. In contrast, linking in Ancona and Zucca's calculus must be done explicitly, by means of the *freeze* operator; imports and exports cannot be combined automatically because they have different name spaces.

5.4.4.1 Self variables versus letrec

Both the m-calculus, and Ancona and Zucca's calculus distinguish between internal and external names. The external name of a member is used by clients of a module to extract definitions, while the internal name is used to write recursive definitions inside the module. External names are part of the interface of a module, and cannot be renamed without changing the module's type; internal names are subject to automatic alpha-renaming, much like a letrec.

Relevance: The DEEP calculus does not distinguish between internal and external names. Instead, each module has a single self-variable; DEEP is similar to ML-style recursive modules in this respect. The advantage of using a self-variable is that it better supports OO-style programming, since a module can pass a reference to itself to other routines.

The advantage of using internal names, on the other hand, is that internal names make it much easier to analyze recursive references within the module. One of the distinguishing features of Wells' m-calculus is that the reduction rules are confluent, and the theory is thus equational [Wells and Vestergaard, 2000].

Wells establishes confluence for the m-calculus by imposing an order on internal

names. It would be very difficult to impose an order on names in DEEP, because our use of self-variables makes it hard to distinguish between internal and external references. As a result, reduction in the surface syntax of DEEP-- is not confluent, and we must flatten the module hierarchy as described in Section 3.5.1.

5.4.5 Late binding for types

Neither Ancona and Zucca's calculus, nor Wells' m-calculus deal with type members within modules. The *Units* proposed by Flatt and Felleisen are defined somewhat similarly, and claim to support type members, but Flat and Felleisen do not provide many technical details on exactly how type members should be handled [Flatt and Felleisen, 1998]. They note that only certain forms of type definitions are permitted, since arbitrary equations would not be contractive.

The mixin modules of Duggan and Sourelis support late binding for both functions and datatype declarations [Duggan and Sourelis, 1996]. Unlike the other module systems described in this section, Duggan and Sourelis explicitly support OO-style implementation inheritance. Functions in a mixin module can forward behavior to derived modules using the inner keyword, which is adapted from Beta. Modules may also give partial definitions of variant data types; when the modules are mixed together, the variants declared in different modules will be merged to generate a single data type. These two mechanisms allow Duggan and Sourelis to solve the expression problem in an elegant way [Duggan and Sourelis, 1996].

Nevertheless, the module system proposed by Duggan and Sourelis is somewhat restrictive. Type members must be datatype declarations (i.e. variant data types); arbitrary type equations are not allowed. Perhaps most importantly, they define mixins only for simple modules that do not contain nested modules; it is not possible to mix large, hierarchically structured modules together.

5.5 Virtual Classes

In object-oriented languages, a class encapsulates a group of interacting methods, along with the data that those methods operate on. Object-oriented inheritance can be used to add new methods, add new data members, or to override existing methods within a class. Many OO languages, including both Java and C++, also allow classes to be nested within other classes. However, the mechanism of inheritance only applies

to methods; it is not possible to override or refine a nested class.

Virtual classes have been proposed for OO languages as a way of easing this restriction. If a language supports virtual classes, then it is possible to refine nested classes using inheritance. The technique is attractive for programming in the large, because inheritance becomes a tool for manipulating whole class hierarchies [Ernst, 2003].

Virtual classes were first introduced in the Beta language, and were initially presented as an alternative to parameterized classes — a.k.a. generics [Madsen and Møller-Pedersen, 1989]. However, Beta was unable to do static typing for virtual classes, and was forced to insert run-time checks. The Beta language also had a major restriction, which is that it was not possible to inherit from a virtual class. Without virtual inheritance, it is not possible to implement *higher-order hierarchies* [Ernst, 2003], which are required to solve the expression problem.

The gbeta language was designed by Ernst as "generalized Beta" [Ernst, 1999b]. It removes the restriction on virtual inheritance, and provides a static type system. Ernst has published a series of papers which shows how virtual classes can be used to solve a variety of perplexing problems in a statically safe way; see [Ernst, 1999a], [Ernst, 2001], [Ernst, 2003]. Unfortunately, the type system for gbeta is complex, and was quite difficult to formalize.

In recent years, virtual classes have enjoyed something of a renaissance, and a variety of approaches have been published for dealing with them. These approaches can be placed into three categories:

1. Virtual classes are members of objects — they are treated like inner classes in Java.
2. Virtual classes are members of classes — they are treated like static nested classes in Java.
3. Virtual classes are members of a class group, or some other kind of module.

The first mechanism is the most powerful, because it means that an unlimited number of different class families can be instantiated at run-time. The first mechanism is also the most difficult to implement, however, because if classes depend on objects, then the type system must be dependently typed. For practical programming it is not clear whether or not the additional flexibility is worth the hassle, so the second two approaches have been pursued as simpler techniques.

5.5.1 Virtual class calculi

The *vc* calculus [Ernst et al., 2006] is a formalization of the gbeta and Caesar languages, and was the first calculus to provide full support for virtual classes as members of objects. The definition of *vc* includes several complications that do not appear in DEEP. First, *vc* is an imperative language with a mutable heap. As such, the theory must keep track of *stable identifiers* — object expressions that do not change over the course of program execution. Second, *vc* implements multiple inheritance by linearization.

Typing in the *vc* calculus is provably decidable, but it gains decidability at the cost of several restrictions. The type system of *vc* is purely nominal. All types in *vc* must be a dependent path of the form $x.l_1.l_2...l_n.C$, where x is a self-variable, and C is the name of a class member. Two types are considered to be equal if they have the same path. The *vc* calculus does not support full dependent types, and thus does not need to compare arbitrary object expressions.

As discussed in Section 4.5.3, *vc* does not support final bindings, it does not support parameterized classes, and it is not possible for inheritance to cross class boundaries. A base class and its derived classes must all be defined within the same enclosing class. These restrictions are a consequence of linearization.

The Tribe calculus is both simpler and somewhat more powerful than *vc*, although its decidability has not been proven [Clarke et al., 2007]. Like DEEP, Tribe uses singleton types to enrich the subtype relation. Like *vc*, a base class and derived classes in Tribe must be declared within the same enclosing class.

Nystrom and Myer's Jx language is an extension of Java which supports the refinement of static nested classes [Nystrom et al., 2005]. In other words, virtual classes are members of classes rather than objects. The most unusual aspect of Jx is the use of *prefix types* instead of path types. Java provides the this keyword as a self-variable for objects, but there is no corresponding keyword that's a self-variable for classes. Prefix types are a way of circumventing this restriction. The J& language further extends Jx with multiple inheritance [Nystrom et al., 2006].

Classboxes have been proposed for Smalltalk as a way of implementing virtual classes and higher-order class hierarchies [Bergel et al., 2005]. Since smalltalk is a dynamically typed language, no type system has been formalized.

The .FJ calculus developed by Igarashi and Saito is an extension of featherweight Java which allows the refinement of static nested classes[Igarashi et al., 2005]. Like

Beta, it is not possible to inherit from a virtual class in .FJ.

One of the earliest attempts at a theory of virtual classes is the one proposed by Bruce, Odersky, and Wadler, which allows groups of classes to be safely refined [Bruce et al., 1998]. However, their mechanism does not scale well — there is no way to define groups of groups, and it is not possible to inherit from a virtual class. This issue is discussed in more detail in [Bruce, 2003]. The Concord language developed by Jolley et. al. is a more recent formulation of class groups which lifts the restriction on inheritance [Jolly et al., 2005].

The idea of "open classes" described in [Clifton et al., 2000] is somewhat similar to virtual classes. An open class is one which is not fully defined at the point where it is declared, and thus can be extended with new fields and methods outside of the original class definition. The C# language now has a similar capability, called *partial classes*.

Open classes differ from virtual classes because they are not nested members of a module or other class. With virtual classes, each extension of a module creates a new class family that is separate from the original, so it is possible to define several different families of classes which have been specialized to do different things. An open class is just an ordinary class which has a definition that is split across multiple sections of code, so it is not possible to create separate families in this way.

5.5.2 Virtual types

Virtual types are somewhat similar in concept to virtual classes. A virtual type is a type alias that can be specialized in subclasses. Because it is just an alias, rather than a full class definition, virtual types are somewhat simpler to deal with than virtual classes. (For one thing, there is no need to consider virtual inheritance.) Virtual types are also less powerful – they are somewhat similar in power to more familiar mechanisms like Java Generics.

Mads Torgersen was one of the first people to present a formal type theory for virtual types that was statically safe [Torgersen, 1998], as well as provide some of the classic motivating examples. Igarashi and Pierce give a formal model of virtual types in a λ-calculus with dependently-typed records [Igarashi and Pierce, 1999].

Odersky's νObj calculus [Odersky et al., 2003] also supports virtual types, and is the basis for the Scala programming language [Odersky and et. al., 2004]. Like the *vc* calculus, νObj is dependently typed, but restricts the use of dependent types to simple

paths. Scala does not support virtual classes per se, but it does support nested classes.

Scala also has an unusual feature, which is that the self-type of a class can be explicitly declared. In an abstract class, the declared type of "self" does not need to be related to the actual definition of self. Any discrepancy between the type of self, and the definition of self, is not resolved until a concrete class is derived from the abstract class. This mechanism allows Scala to emulate virtual classes to a large degree, by using virtual types as the declared self-type of nested classes [Odersky and Zenger, 2005]. The main limitation of the Scala approach is that nested classes must be mixed together by hand, a process that is both complex and error-prone [Zenger and Odersky, 2005].

5.6 Aspects and Features

Aspect-oriented programming (AOP) is a somewhat radical approach to modularity that has been proposed in recent years. Since the idea was initially invented by Kiczales in 1997 [Kiczales et al., 1997], its popularity has grown rapidly. Not only is there now a yearly conference on aspects, but papers on AOP regularly appear in conferences on OOP and generative programming as well. This section offers a brief introduction to the concepts of AOP; it is not a survey of research in the field.

The goal of AOP is to provide mechanisms for *separation of concerns*. In general, the idea behind separation of concerns is much the same as traditional ideas of modularity. The goal is to factor a program into separate modules, called *aspects*, where each aspect governs a particular "concern" the program must deal with.

AOP differs dramatically from previous work on modules because of the way in which factorization is done. Certain concerns are *cross-cutting concerns*, which means that they cannot be easily factored out using traditional modules. The quintessential example of a cross-cutting concern is logging code, which records the transactions that are made to a system. Every method that performs a transaction must contain a line of code that writes a message to the log file. Ideally, it should be possible to factor the code that does logging into a separate module from the code that performs the transactions. This is hard to do in Java, because the logging code is scattered through a number of different classes and methods. However, aspect languages such as AspectJ provide mechanisms for performing such factorization [Kiczales et al., 2001].

Aspects are relevant to the ideas in this thesis because the expression problem is an excellent example of a cross-cutting concern (see Section 4.8.2). If an interpreter is

written in an object-oriented style, then a new operation *cross-cuts* the existing class definitions. If the interpreter is written in a functional style, then a new variant cross-cuts the existing function and type definitions.

The expression problem thus illustrates another important concept from AOP: the *tyranny of the dominant decomposition* [Peri Tarr, 1999]. No matter how a program is factored, there will always be some concerns that cross-cut the primary modular structure.

5.6.1 Aspect-weaving

An AOP program is structured as a base program, and a number of aspects. Each aspect specifies a code transformation, which modifies the base program by inserting code at certain specified points. An aspect-weaver then weaves all of the aspects together to create a final program. Weaving is usually implemented as either a source-code transformation, or a byte-code transformation [Kiczales et al., 2001].

An aspect can add new data members and methods to existing classes, or add new code to existing methods. In this respect, aspects are somewhat similar to inheritance with virtual classes. Aspect weaving differs from inheritance because it is possible to use patterns to quantify over many classes or methods [Kiczales et al., 2001]. In the case of logging code, it is possible to add a statement to all methods that have a particular type signature or name that matches a particular regular expression.

Although aspects are enormously flexible, they suffer from the same theoretical issues that affect virtual classes and mixin modules. Since aspects are code transformations, the order in which aspects are applied matters. However, aspects are not explicitly applied, so it can be difficult to control ordering; the situation is somewhat similar to the linearization problem of mixins. Also like mixins, it is possible for two aspects to make incompatible modifications, such as adding two fields with the same name but different types.

The most serious problem with existing aspect languages is that it is very difficult to type-check aspects separately. Modular type-checking of mixins is already difficult (see section 4.4) and the use of quantifiers does not make things any easier. In practice, aspect compilers make heavy use of code generation; type-checking is not done until after aspect-weaving [Kiczales et al., 2001].

5.6.2 Feature-oriented programming

Feature-oriented programming (FOP) predates AOP, and is very similar to AOP in many ways [Batory and OMalley, 1992] [Batory et al., 2003] [Batory et al., 2004]. The term "feature-oriented programming" is relatively recent; the basic ideas have also been called *multi-dimensional separation of concerns* [Peri Tarr, 1999] and *mixin layers* [Smaragdakis and Batory, 2002].

A feature, like an aspect, is a slice of program behavior that cross-cuts the usual module structure. FOP tools factor these slices into separate modules, and then compose the various features together using a source-to-source transformation tool.

Features differ from aspects because they take a more disciplined approach to composition [Lopez-Herrejon et al., 2006]. There are no quantifiers, so feature composition closely resembles OO inheritance, except applied on a much larger scale. Whereas the emphasis in AOP is on making transformations as expressive as possible, the emphasis in FOP is on making transformations predictable. The features that are to be composed, and the order in which they are composed, is specified explicitly by a feature equation. FOP composition tools also provide mechanisms for specifying dependencies and interactions between features [Batory et al., 2003] [Liu et al., 2005].

Some examples of feature interactions are as follows. A feature F may depend on G. Features H and I may be mutually incompatible — only one should ever be applied within the same program. If features F and H are both applied in the same system, then feature J is also required. The last example can be seen in the expression problem; given a new class Times, and a new method toString, a *feature derivative* [Liu et al., 2005] must be applied to supply the definition of toString for Times.

Relevance: The DEEP language has been heavily inspired by feature oriented programming. In many respects, DEEP is a type theory for features — the way in which modules are composed is essentially identical to FOP composition tools like AHEAD.

5.7 Prototypes and delegation

The idea of using prototypes and delegation, rather than classes, as a means of structuring OO programs was first suggested by Lieberman [Lieberman, 1986]. The idea was later used in the design of the Self language, which remains the most well-developed implementation of the concept [Ungar and Smith, 1987].

Self is a dynamically-typed language, and relatively little work has been done on the type theory of delegation, in part because prototypes were initially advertised as an alternative to classes, and hence an alternative to the restrictions of statically-typed class-based programming. A simple nominal type system based on prototypes was implemented by Blascheck [Blaschek, 1991], and the relationship between delegation and subtyping was studied by Fisher and Mitchell [Fisher and Mitchell, 1995]. Fisher and Mitchell were the first to point out the difference in typing requirements between using an object, and inheriting from an object.

A much more extensive study of prototypes was done by Kniesel as part of his PhD thesis [Kniesel, 2000]. Kniesel provides a detailed discussion of many issues relevant to the type safety of prototype languages, but does not present a formal calculus in the style developed here.

The generic wrappers proposed by Buchi and Weck allow one class to delegate behavior to a different class [Buchi and Weck, 2000]. Since delegation links are established at run-time, name clashes must also be detected at run-time.

Osterman's dynamically composable delegation layers [Ostermann, 2002] also use delegation to compose modules at run-time. Unlike generic wrappers, delegation layers can affect whole class hierchies, thus combining the ideas of virtual classes and delegation.

Abadi and Cardelli's object calculus is also related to protoype languages [Abadi and Cardelli, 1996a]. Although Abadi and Cardelli do not use delegation, their calculus is purely object-based; there are no classes. However, their object calculus does use types, and it clearly distinguishes between types and objects.

5.8 Generative Programming

Generative programming [Czarnecki and Eisenecker, 2000] and meta-programming [Kiczales et al., 1991] are techniques that involve manipulating other programs as data. Generative programming is by no means a new technique. Every compiler is a generative program, as are many commonly-used tools like lex and yacc [Levine et al., 1992]. The idea of programs-as-data is also a defining characteristic of the Lisp programming language [Guy L. Steele, 1990].

Most software today is written by hand. The goal of generative programming is to automate the software creation process to some extent, so that human programmers can concentrate on those areas of software design that really require human input, while

offloading certain tedious and error-prone tasks to automatic routines. Historically, the advent of good quality compilers meant that programmers seldom had to work directly with assembly language, and the ensuing improvements in programmer productivity are well-recognized. Today, work on program generation can be broadly classified into two areas: (1) improvements to general-purpose languages, and (2) creation of domain specific languages.

General purpose languages continue to evolve, as programmers seek ever more powerful forms of abstraction. Aspect-oriented programming and feature-oriented programming are two primary examples of general-purpose generative techniques. The goal in both cases is to extend an existing programming language (typically Java) with new forms of abstraction. The extended language is "compiled" down to ordinary Java, which in turn is compiled to byte-code, which is compiled to machine code in a cascade of program generation steps.

Domain specific languages (DSLs) provide a set of abstractions that are custom-tailored to a particular problem domain. Classic examples are database query languages like SQL, parser generators like lex and yacc [Levine et al., 1992], or self-optimizing numeric libraries like Blitz++ [Veldhuizen, 1998]. In most cases, these too must be "compiled" to a general-purpose language.

Generative programming is hard because although it is easy to store programs as data, it is hard to manipulate that data in a meaningful way. Primitive program generators, such as the C preprocessor macros [Ellis and Stroustrup, 1990], manipulate code in the form of simple ASCII text. There is no guarantee that such manipulations are even syntactically valid, to say nothing of semantics. Lisp macros are an improvement because they operate on abstract syntax trees, and are thus guaranteed to produce a syntactically valid program [Guy L. Steele, 1990]. Nevertheless, it is difficult to ensure that the code generated by a macro is correct, or even that it is well-typed.

Very sophisticated program generators, such aspect-weavers and feature composition tools, suffer from the same problem. Type-checking of an AOP or FOP program happens after code generation. Parser generators like lex and yacc also delay type-checking until after code generation, as do C++ templates. This delay in type-checking is a serious problem, because it is very difficult for human programmers to either debug or understand computer-generated code.

Well-typed programs may still contain errors, of course, but if a program generator cannot even show that generated code is well-typed, which is a relatively simple property, it will be even harder to show that the code is correct in other ways.

5.8.1 Partial evaluation

Partial evaluation is a particularly simple approach to program generation which neatly solves the problem of correctness. A partial evaluator transforms an input program by applying the standard reduction rules of the language in question. In a statically typed language, reducing an expression will generate an equivalent expression with the same type. If the initial program is well-typed, then all code generated by the partial evaluator is also guaranteed to be well-typed. We regard this as a major advantage of partial evaluation over other forms of generative programming.

The tricky part of implementing a partial evaluator lies in choosing which expressions to reduce. If the evaluator were to simply reduce all expressions, then it would not be "partial"; it would be a (rather inefficient) interpreter. For each expression in a program, the partial evaluator must decide whether to reduce the expression at compile-time, or defer evaluation to run-time, in which case the expression will be left as a *residual* in the compiled code. There are two ways to make this decision: *online* evaluation, and *offline* evaluation [Jones et al., 1993] [Jones and Glenstrup, 2002].

Online partial evaluators decide what to reduce on the fly. In general, any expressions involving constant arguments, such as (3 + 4) can be reduced immediately, while those involving unknown variables, such as $(x + 4)$ must be deferred to run-time. (A real-world program usually obtains some of its data from files or user-input, so many variables will have unknown values at compile-time.) However, a simplistic strategy of "reduce everything possible" is too agressive, and will typically fail to terminate in the presence of recursive functions; section 3.6.3 gives some examples. Thus, the challenge of online evaluation is choosing a strategy that is aggressive enough to give real optimization benefits, but not so aggressive that it fails to terminate [Jones et al., 1993] [Jones and Glenstrup, 2002].

Offline partial evaluators perform partial evaluation in two phases: a *binding-time analysis* phase, followed by an *specialization* phase. Binding time analysis (BTA) is a phase somewhat similar to type inference. Each subexpression in a program is labeled as either "static", meaning evaluate at compile-time, or "dynamic", meaning evaluate at run-time. The specialization phase then reduces a program according to the annotations supplied by BTA.

Binding time analysis has several advantages. One advantage is that BTA makes it possible for the programmer to see exactly which computations will be perfomed by the evaluator, merely by looking at the annotations on the original source code. This

capability is useful because it is often necessary to write "binding time improvements" [Jones, 1995] which restructure the code in order to improve the effectiveness of partial evaluation. Moreover, it is often easier to ensure termination in an offline evaluator [Jones and Glenstrup, 2002].

Binding time analysis is also potentially much more efficient. No decisions have to be made during the specialization phase. In practice, this means that annotated code can essentially be compiled for efficient partial evaluation. If a great deal of code is being generated, or if the evaluator employs run-time code generation [Taha, 2003], then this is a significant gain. It is also important for the practical design of self-applicable partial evaluators, which have been studied as a way of constructing compiler generators [Jones et al., 1993] [Futamura, 1999].

5.8.1.1 Binding time analysis

The primary disadvantage of binding time analysis is that because all decisions are made ahead of time, it is hard for the evaluator to make use of information that may result late in the evaluation process. As an example, consider the problem of doing binding-time analysis on the humble map function, shown here in Haskell:

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = (f x) : (map f xs)
```

BTA must answer the following question: should (f x) be a static computation, or a dynamic computation? This question is hard to answer because it involves reasoning about higher order functions, partially static data structures, and polyvariant binding-time analysis.

The expression (f x) can only be static if f is static, but because map is a higher-order function, the value of f is only known at the call site for map, not the point at which map is declared. Moreover, knowing that f is static is not enough; we need more detail about the binding time analysis of f. Can f be evaluated with a dynamic argument, or does it require a static argument? If it requires a static argument, then is x static?

In order to determine whether x is static, it is necessary to extend binding time analysis to data types as well as functions. A list may be fully static, fully dynamic, or somewhere inbetween. The list [1, 2, 3] is fully static. It is also possible to have a static list with dynamic elements, e.g. [x, y, z], a static list with a mixture of static and dynamic elements, e.g. [1, 2, z], or even a list with a static head but a dynamic tail, e.g. 1:2:zs. (The variables in these examples represent dynamic expressions).

The lists [1, 2, 3] and [x, y, z] are regular, because all elements are either static or dynamic. The other two lists are irregular. In general, binding time analysis can deal with regular data structures, but not irregular ones. An good offline partial evaluator would generate the following results for different expressions with map:

```
map (∗2) [1, 2, 3]   ⟶   [2, 4, 6]
map (∗2) [1, 2, z]   ⟶   [1∗2, 2∗2, z∗2]
map (∗2) 1:2:zs      ⟶   map (∗2) 1:2:zs
```

If the list contains a single dynamic element (i.e. z), then all applications of (f x) in map must be judged as dynamic, thus yielding a list which contains expressions (e.g. 2∗2) that could clearly have been further reduced. If a list contains a dynamic tail (i.e. zs), then it must be labeled as "fully dynamic", and the partial evaluator will not reduce map at all.

Moreover, in order to achieve results that are even as good as those shown above, an offline partial evaluator must be able to perform *polyvariant* binding-time analysis [Jones et al., 1993] [Christensen et al., 2000]. Since map is a commonly used function, there is no one binding time that can be assigned to it. The evaluator must thus assign many different binding times to it, and then select the best version depending on context.

An online partial evaluator does not need to deal with these complexities, and can potentially perform the following reductions:

```
map (∗2) [1, 2, 3]   ⟶   [2, 4, 6]
map (∗2) [1, 2, z]   ⟶   [2, 4, z∗2]
map (∗2) 1:2:zs      ⟶   2:4:(map (∗2) zs)
```

Because online and offline evaluation have different strengths, a good deal of work has focused on either making offline evaluation more flexible [Christensen et al., 2000], making online evaluation more efficient [Sumii and Kobayashi, 1999], or otherwise attempting to combine the two techniques.

5.8.2 Multi-level and multi-phase languages

Much of the early work on partial evaluation was done in Lisp and Scheme, which are untyped languages. Both online and offline partial evaluation can be used in typed languages as well, but a third variation is also possible: multi-level languages.

In a multi-level language, such Nielson and Nielson's two-level λ-calculus [Nielson and Nielson, 1992], information about the binding time of an expression is attached to

the type of that expression. For example, a compile-time integer expression would have type Int, while a run-time expression would have type $\underline{\text{Int}}$: the underline denotes a dynamic expression. One advantage of including binding-time information in the type is that the technique scales naturally to polymorphic data types — [Int] is a static list of static values, e.g. $[1,2,3]$, while $[\underline{\text{Int}}]$ is a static list of dynamic integer expressions (e.g $[x,y,z]$), and $\underline{[\text{Int}]}$ is a fully dynamic list.

A simple multi-level language is somewhat rigid, because it corresponds to monovariant binding time analysis. Danvy's technique of type-directed partial evaluation [Danvy, 1996] extends a two-level language with *reify* and *reflect* operators that cast expressions from dynamic to static and vice versa, thus improving the flexibility of the evaluator.

MetaML and Ωmega are multi-stage languages [Taha and Sheard, 2000] [Taha, 2003] [Sheard, 1994] [Sheard, 2005]. Unlike partial evaluation, which typically defines only two stages (compile-time and run-time) MetaML supports any number of stages. Like the two-level lambda-calculus, expressions in MetaML either have type T, meaning "evaluate in the current stage", or type $code(T)$, meaning "evaluate in the next stage". Any expression can be quoted to label it as "next stage". Quoted expressions may have unquoted subexpressions; these will be evaluated in the current stage to yield a result of type $code(T)$, and then spliced into the quoted expression.

In addition to the quote and unquote annotations, which are essentially the same as underlines in an ordinary two-level λ-calculus, MetaML has a *run* command, which takes a term of type $code(T)$, and evaluates it in the current stage to yield a result of type T. This command is what allows a program to be truly multi-stage: the code executed by *run* may also use quote, unquote, and run to build and execute code.

MetaML also differs from traditional partial evaluators because it uses run-time code generation rather than compile-time code generation. Run-time code generation is potentially much more powerful, since a program can customize its behavior based on user input.

Staging, as found in MetaML and Ωmega, has two main limitations. Unlike traditional binding-time analysis, all annotations must be written by the programmer rather than inferred, which is a rather high burden. Second, staging corresponds to monovariant binding time analysis. A function like map that may be called in different ways must be written multiple times, using a different syntax for each version; this is a significant impediment to code reuse.

5.8.3 Template Haskell

Template Haskell is unique as a generative programming language because it does not fit into any of the categories (online, offline, or multi-stage) mentioned above [Sheard and Peyton Jones, 2002]. Like MetaML, Template Haskell uses quote, unquote, and run. Unlike MetaML, however, Template Haskell is not guaranteed to produce well-typed programs. Like many other generative programming technologies, Template Haskell performs type-checking after code generation.

Dynamic expressions in Template Haskell are untyped; they have type Expr, rather than type Expr(T). It is thus possible for a code generator to produce ill-typed code. However, it is not possible to run ill-typed code. A dynamic expression must be type-checked before it can be reflected to a static expression. The process of compiling a program in Template Haskell will alternate between code generation and type-checking as needed.

5.9 The expression problem

The expression problem is a well-known problem that has been discussed extensively in the literature, dating back to [Reynolds, 1975]. Cook identifies the basic problem when comparing object-oriented systems to abstract data types [Cook, 1990]; he mentions that OO systems and ADTs support different mechanisms for extension. As such, the problem is sometimes called the "extensibility problem" [Zenger and Odersky, 2001].

The name "expression problem" was coined by Wadler [Wadler, 1998]. Wadler formulated a precise description of the problem as involving an extensible interpreter in particular, and outlined a set of requirements that a solution to the problem should satisfy. Many "solutions" in the literature do not meet all of Wadler's requirements; in particular, Wadler specifies that each extension should be both statically typed and separately compiled.

Design patterns can be used to ameliorate the expression problem to some degree. The visitor design pattern [Gamma et al., 1995] allows operations on a datatype to be defined by cases, thus mimicking the functional programming style in an OO language. Records of functions can also be used to mimick the OO programming style in a functional language. However, simple patterns such as these only change the axis (operations versus data variants) on which extensions can be added; they do not allow

simultaneous extensions on multiple axes.

Krishnamurthi et. al. provides a detailed discussion of the problem, and proposes a more sophisticated design pattern which solves it [Krishnamurthi et al., 1998]. The essence of their approach is to use factory methods as "virtual constructors" for classes. Factory methods can serve as a substitute for virtual classes to some degree by ensuring that a more specific class is constructed at the appropriate places. However, the solution in [Krishnamurthi et al., 1998] requires dynamic type casts and is thus not statically typed. Moreover, the "extensible visitor pattern" that they propose is complex; so complex, in fact, that they propose adding special syntax to the language to lower the burden on programmers.

Torgersen suggests no less than four alternative approaches in [Torgersen, 2004], which use a combination of design patterns, Java wild cards, and F-bounded polymorphism. Torgersen notes that the solutions using F-bounded polymorphism, while effective, are very complex and difficult to write.

Swierstra presents a solution to the expression problem in Haskell in [Swierstra, 2008]. Swierstra's solution does not use ordinary Haskell data types, or ordinary pattern matching on data types. Instead, he implements a set of type constructors which allow new data types to be constructed as needed, and then uses Haskell's powerful type class mechanism to implement operations over those data types. Like Torgersen's solutions in Java, Swierstra's solution in Haskell requires a sophisticated coding protocol.

We argue in this thesis that all solutions that require a complex design pattern or coding protocol are flawed for two reasons. First, they require the programmer to specifically anticipate the need for extension before program development even begins. Second, a complex protocol is both difficult to design, and difficult to use. We feel that languages should provide mechanisms that allow programmers to structure code in whatever way seems most natural, without sacrificing extensibility.

Extensible algebraic datatypes [Zenger and Odersky, 2001] is an example of a language mechanism that improves extensibility. It combines variant data types with object-oriented inheritance. A class may declare named constructors that define different variants, and the number of variants can be extended in subclasses. For example:

```
class Term {
  case Lit(int i);
  case Plus(Term t, Term u);
}
```

```
class MultTerm extends Term {
  case Times(Term t, Term u);
}
```

Extensible algebraic datatypes solve the expression problem in a very similar way to the FP-style solution in DEEP described in Section 4.8.4. However, their usefulness beyond the expression problem itself is more limited. They cannot be used to encode full-scale extensible class hierarchies as described in [Ernst, 2003].

In object-oriented languages, virtual classes provide the most elegant and widely-studied solution to the expression problem. A solution in gbeta is described by Ernst in [Ernst, 2003], but a similar solution could be written in several of the systems listed in section 5.5.1. However, not all virtual class calculi are strong enough. Ernst's solution to the expression problem requires a subclass that inherits from a virtual class, so his solution does not work in calculi that do not allow virtual inheritance; this limitation is discussed by Bruce in [Bruce, 2003].

The Scala language does not allow virtual inheritance, but it does support virtual types and explicit self-types; a solution to the expression problem using these techniques is described in [Zenger and Odersky, 2005]. The solution in Scala is more complex than the one in gbeta because without true virtual classes, nested classes must be mixed together by hand, a process that is both tedious and error-prone.

In functional languages, Duggan's mixin modules can also be used to solve the expression problem [Duggan and Sourelis, 1996]. Mixin modules are in many ways the functional programming equivalent of virtual classes. The types in question are recursive variant data types instead of recursive OO classes, but they are refined in a very similar fashion.

Nakata and Garrigue have also written a solution to the expression problem using recursive ML modules with applicative functors [Nakata and Garrigue, 2006]. Their solution implements delayed binding for types (see Section 3.2.2.1); it uses a signature to hide the type definition, uses a functor to abstract over the self-variable of the module, and then takes the fixpoint of the functor. The main limitation of this approach is that the data type must be completely opaque (i.e. fully hidden). Without a strong notion of subtyping, it is not possible to express the fact that the new type is a refinement of the old.

Finally, both aspect-oriented programming [Kiczales et al., 2001] and feature-oriented programming [Batory et al., 2003] [Peri Tarr, 1999] can be used to solve the expression problem, using the same programming technique that is employed by vir-

tual classes. However, because AOP and FOP are generative programming techniques, extensions cannot be type-checked or compiled separately.

Neither of the solutions to the expression problem that we show in DEEP are particularly unusual. The OO solution matches the one given in [Ernst, 2003] using virtual classes, and our FP solution is similar to ones based on extensible algebraic data types [Zenger and Odersky, 2001] and mixin modules [Duggan and Sourelis, 1996]. What is unusual about our solution is that we can further extend it to do tag elimination as well; this is not possible in any of the languages described above.

5.10 The tag-elimination problem

The tag elimination problem was identified by Jones in [Jones et al., 1993] as an outstanding problem in the field of partial evaluation for typed languages. One of the earliest goals of partial evaluation research was to automatically generate a compiler for a language, given an interpreter for that language [Futamura, 1999].

When evaluating the effectiveness of partial evaluation techniques, one would like to show that the resulting compiler is "optimal", in the sense that all interpretive overhead has been removed. However, if the source and target languages are different, then defining what "optimal" means is not easy. It is hard to tell whether certain parts of compiled code are "interpretive overhead", or legitimate artifacts of the translation.

Jones thus defined optimality by setting the source and target languages to be the same [Jones et al., 1993]. Let L be a language which is equipped with a partial evaluator. An interpreter is written in L, and interprets expressions in L^-, where L^- is a subset of L. Partial evaluation is optimal if for any abstract syntax tree representing an expression in L^-, partially evaluating the interpreter with respect to the AST will generate an expression in L which is identical to the original AST in L^-. Since L^- is a subset of L, the translation can be exact. L^- is commonly set to be the simply typed λ-calculus in most studies, since the λ-calculus is a common subset of most functional languages.

Generating optimal compilers in an untyped language was achieved early on. However, in a typed language, the interpreter must typically pack results into a universal data type of some kind, as shown in Chapter 1, Figure 1.5. Results are "tagged" with their run-time type, and neither the tags nor the branches on tags are removed by partial evaluation. In an untyped language, a universal data type is not required, and so the problem never arises.

There have been a number of proposals in the literature for solving the tag elimination problem, which can be broadly classified into two categories: those based on very expressive type systems like dependent types or GADTs, and those based on type inference.

5.10.1 Type specialization

Hughes' type specialization is one of the first mechanisms that was proposed to solve the tag elimination problem [Hughes, 1996] [Danvy, 1998]. Type specialization is not really partial evaluation per se. Instead, it exploits the type inference system to achieve an effect that is much like partial evaluation.

The main specialization judgement is $\Gamma \vdash e : T \hookrightarrow e' : T'$, which takes an expression e of type T, and produces a residual (i.e. partially evaluated term) e' of type T'. This judgement is interesting because it combines evaluation and typing; as the term e is specialized, its type will change to reflect the specialization. This change in type is what allows tags to be eliminated; the type of a tagged expression will be specialized to one in which the tags are known.

Type specialization uses a two-level λ-calculus which has been extended with singleton types. Static values like 3 and true are types which are inhabited by a single object $\bullet$. Because such values are types, they can be propogated by unification and type inference.

Unlike ordinary partial evaluation, type specialization is able to propogate static values without actually reducing or inlining function applications. If type of a function argument x is specialized to a singleton, then the type of expressions involving x within the body of the function will also be specialized. The result of specialization is a residual program which includes $\bullet$ in places where computations were done statically. The type specialization pass is thus followed by a dead-code elimination pass to remove singleton values.

Relevance: Type specialization is not directly related to DEEP, because DEEP is not a two-level language, and does not perform type inference. However, some of the concepts are quite similar, in particular the use of values as singleton types.

5.10.2 Tag elimination through type inference

Carrette et. al. have proposed a different way of using type inference to perform tag elimination [Carette et al., 2007]. Their approach is unique in that it does not require

an advanced type system of any kind — it can be easily implemented in standard Haskell or ML. Rather than using the built-in data types of Haskell and ML, Carrette et. al. write the data type constructors directly as functions which perform a fold operation on the AST. This encoding allows them to write an interpreter that does not use a universal data type.

The main disadvantage of the approach in [Carette et al., 2007] is that it bypasses the standard mechanism provided by Haskell and ML for pattern matching over data types, so programming in their framework is more difficult. As mentioned in Section 5.2.2.2, pattern matching is the most complex part of type-checking for GADTs, so it is not entirely surprising that an encoding that avoids pattern matching can obtain the benefits of GADTs without the complexity of type-checking.

5.10.3 Dependent types and GADTs

The most popular way of solving the tag-elimination problem in the literature is to use a very expressive type system, such as dependent types or GADTs.

This approach was initially investigated in [Taha and Makholm, 2000] and [Taha et al., 2001]. Taha and Makholm did not initially rely on the type system, but instead introduced tag elimination as a post-processing step which literally removes the tags from the source code. Taha and Makholm found that eliminating tags in this way produces a program that is type-safe if and only if the program being interpreted is well-typed. It is not safe to eliminate tags if the input program is ill-typed; the type tags are essentially a form of dynamic typing.

This work inspired solutions to the tag-elimination problem that are based on dependent types [Pašalić et al., 2002]. In a language with dependent types, it is possible to write the interpreter in such a way that it only accepts well-typed input [Augustsson and Carlsson, 1999]. Exploiting the Curry-Howard isomorphism, a well-typed interpreter accepts three arguments: an environment, a term, and a proof that the term is well-typed in the given environment.

The problem of the well-typed interpreter is one of the main motivating examples that inspired the development of GADTs (see Section 5.2.3). The full power of dependent types is not necessary for the simply typed λ-calculus; GADTs will suffice. Rather than passing a proof of well-typedness as an argument to eval, GADTs allow a well-typed interpreter to be defined by associating a type index with constructor. The following unpublished example was provided by Philip Wadler:

```
data Id e a where
  Z :: Id (a,e) a
  S :: Id e b -> Id (a,e) b

data Term e a where
  Lit  :: Int -> Term e Int
  Var  :: Id e a -> Term e a
  Fun  :: Term (a,e) b -> Term e (a -> b)
  Plus :: Term e Int -> Term e Int -> Term e Int
  App  :: Term e (a -> b) -> Term e a -> Term e b
```

The solution to the tag elimination problem in DEEP (see Section 4.8.9) broadly follows the pattern shown above. Our solution in DEEP differs from the Haskell solution only where variables are concerned. In DEEP, variables are indexed by an ordinary integer, whereas the Haskell solution declares Id as a special data type for natural numbers. The use of Id is necessary because Haskell does not have dependent types.

5.10.4 Fairness

All of the approaches to tag elimination discussed above work by using the type system in various ways to ensure that the interpreter only accepts well-typed programs. Passing an ill-typed argument to the intepreter will be flagged as a static type error.

Makholm has designed a tag-elimination system that eliminates tags for well-typed expressions, but leave them in place for ill-typed expressions [Makholm, 2000]. He refers to this property as "fairness". This approach essentially corresponds to the technique of *soft typing* [Cartwright and Fagan, 1991], a form of type inference used in dynamically-typed programs, rather than full static typing.

Makholm's tag-eliminator is implemented as post-processing phase, and is not integrated with the type system. The question of how to implement an interpreter that does soft typing within a statically typed language is an interesting one, and remains an open problem.

Chapter 6

Conclusion

And they lived happily ever after... [except that] Inigo's wound reopened, and Westley relapsed again, and Fezzik took the wrong turn, and Buttercup's horse threw a shoe...

— William Goldman, The Princess Bride

This thesis introduces a novel approach to type theory, which we call *pure subtype systems* (PSSs). Pure subtype systems differ from traditional type systems in two ways. First, they do not distinguish between types and objects; a uniform syntax and semantics is used for all terms. Second, subtyping, rather than typing, is the primary relationship between terms.

The main contributions of this thesis can be divided into two parts: the basic theory of pure subtype systems, and a practical application of that theory to virtual classes and mixin modules.

On the theory side, we have explored both the theory and meta-theory of pure subtype systems within the context of System $\lambda_{\triangleleft}$. As it turns out, fundamental properties such as type safety are very difficult to prove. We presented a partial proof, along with a detailed discussion of why a full proof is difficult.

On the practical side, we developed a formal calculus called DEEP that can handle mixin modules and delegation, and we have a working implementation of that calculus. We have verified that DEEP is capable of handling many difficult typing situations that have been discussed in the literature, including two well-known problems: the expression problem, and the tag-elimination problem. Using DEEP, we are able to solve both problems simultaneously, which has never been done before.

6.1 Summary of theoretical results

Bits and pieces of the ideas that underly pure type systems can be found elsewhere in the literature. The formal foundation of our approach is the theory of higher-order subtyping with bounded quantification, as found in System $F^{\omega}_{\leq}$ [Steffen and Pierce, 1994] [Compagnoni and Goguen, 2003]. The syntax of types and objects is largely unified in Barendregt's pure type systems [Barendregt, 1992], with the exception of the λ and Π binders, which were unified later by several people [de Groote, 1993] [Kamareddine and Nederpelt, 1996] [Kamareddine, 2005]. The unification of typing and subtyping was the basis for Cardelli's power types [Cardelli, 1988] [Aspinall, 2000].

We have taken these ideas, and put them together to construct a theory of subtyping that is entirely independent of typing. Unlike the systems mentioned above, pure subtype systems have no typing relation. This thesis is the first formal study of subtyping on its own. Alternative titles that we considered for the thesis were: "Types and typing considered unnecessary for type theory", and "Subtyping: a substitute for typing".

Our interest in subtyping stems from the fact that it serves as an excellent model of inheritance, both for object-oriented languages, and for advanced module systems. When a subclass inherits from a superclass, it inherits two things: the interface of the superclass, *and the implementation*. Subtyping is usually considered to be a relationship between types, which means that it only models interface inheritance, and not implementation inheritance.

We were inspired by the delegation model used in prototype-based languages like Self [Ungar and Smith, 1987] to view inheritance as a relationship between objects rather than classes. The delegation model explicitly recognizes the fact that both interface and implementation are inherited. Although it is undoubtedly possible to capture the semantics of delegation within a traditional type system, we feel that if one takes delegation seriously then subtyping should be defined as a relationship between objects rather than types.

Inheritance between objects is particularly useful where modules are concerned. Modules are similar to classes in many ways, but in practice they are used somewhat differently. Classes are types, which means that in most programming languages, they cannot be used directly. Only objects can be used in general-purpose computations, so a class must be instantiated to yield an object before it can be used. Modules differ from classes because they are not types, and thus do not have to be instantiated before use. New modules are constructed by directly linking other modules together. We feel

that delegation is a good way to model inheritance between modules.

The three formal calculi presented in this thesis explore these ideas in detail. The DEEP-- and DEEP calculi demonstrate that pure subtype systems can be used to construct a practical theory of mixin modules that is based on delegation between objects. System $\lambda_{\triangleleft}$ demonstrates that these ideas are not restricted to modules and inheritance, but can be interpreted with the larger context of type theory in general.

6.1.1 System $\lambda_{\triangleleft}$

System $\lambda_{\triangleleft}$, which was introduced in Chapter 2, captures the basic concepts of pure subtype systems in the form of a typed λ-calculus. We used System $\lambda_{\triangleleft}$ as a foundation for exploring the basic theory and meta-theory of PSSs.

As a type theory, pure subtype systems are very expressive; subtyping really can be used as a substitute for typing. We demonstrated that the type judgements in both System $F_{\leq}$ and System $\lambda*$ (the pure type system with $* : *$) can be embedded as subtype judgements in System $\lambda_{\triangleleft}$. As mentioned before, the main disadvantage of pure subtype systems is the complexity of the meta-theory.

There are several reasons why the meta-theory of subtyping is so difficult. First and foremost, subtyping is transitive, whereas typing is not. Transitivity elimination is a difficult part of the meta-theory for any type system which includes higher-order subtyping.

In System $\lambda_{\triangleleft}$, there is also a circular dependency between subtyping and well-formedness; subtyping is only defined over terms that are well-formed, but well-formedness depends on subtyping. A similar circular dependency is found in theories of subtyping elsewhere in the literature, as discussed in Section 5.3.2. As is common practice, we unraveled the dependency by introducing an algorithmic definition of subtyping which is defined syntactically over all terms, not just the well-formed ones.

Unfortunately, there is another circular dependency in the meta-theory that we were not able to unravel. The substitution lemma requires transitivity, but the proof that transitivity is admissible requires the substitution lemma. This circularity is a consequence of combining higher-order subtyping with bounded quantification.

Although there are several type systems in the literature that include higher-order subtyping, only one allows bounded quantification on type operators: Compagnoni and Goguen's formulation of System $F^{\omega}_{\leq}$ [Compagnoni and Goguen, 2003]. Their technique of typed operational semantics relies on the fact that the language of types

in System $F^{\omega}_{\leq}$ is strongly normalizing. Their technique thus cannot be used in theories that are not strongly normalizing, such as the ones that we consider in this thesis.

We have developed a new proof technique for dealing with transitivity that is based on a confluence property, rather than strong normalization. Confluence is a well-understood property that has been studied at length in the literature. By formulating the problem in terms of confluence, it is possible to leverage existing proof techniques, such as Van Oostrom's technique of confluence by decreasing diagrams [van Oostrom, 1994].

We formulated the subtype relation as an abstract reduction system with two reduction relations, and showed that transitivity elimination follows from commutativity of the underlying reductions. Unfortunately, although we believe that this is a promising way to approach the theory, we were only able to prove that the reductions commute locally, rather than globally. A full proof of global commutativity will require a stronger inductive hypothesis than we have thus far been able to devise.

Although we were not able to prove type safety, we did develop practical type-checking algorithms. These algorithms are described in Chapter 2, and they have been implemented within the current version of the DEEP interpreter.

6.2 Summary of practical applications

Chapter 3 introduced the DEEP-- calculus, which demonstrates how the ideas of pure subtype systems can be applied to build a type theory for first-class recursive modules. The semantics of DEEP-- follow the delegation model used in Self, so modules inherit directly from other modules.

Modules in DEEP-- have the following characteristics:

- Modules may contain both types and objects.
- Modules are recursive. They support mutually recursive type and object definitions.
- Modules are hierarchical. They may contain nested modules.
- Modules are extensible. Using inheritance, it is possible to add new definitions, override existing definitions, or extend nested modules. Both type and object members use late binding.
- Modules are first-class objects.
- Modules support separate compilation. A field $l : t = u$ will hide the implementation u behind the interface t.

- Modules are translucent. Final bindings allow the precise definition of any member to be exposed in the interface.

These characteristics are strong enough that DEEP-- is able to handle many interesting typing situations. The ability to extended nested modules is particularly important, and is called *deep mixin composition*, from which DEEP derives its name.

We showed how modules can be used to encode virtual types [Torgersen, 1998], family polymorphism [Ernst, 2001], type classes [Wadler and Blott, 1989], and associated types [Chakravarty et al., 2005]. However, DEEP-- is not strong enough to handle the expression problem or mixin modules. It is limited because the parent of a module must be statically known; it is not possible for a module to inherit from a variable or from a virtual class. This limitation prevents DEEP-- from encoding higher-order class hierarchies [Ernst, 2003].

Recursion in DEEP-- is potentially problematic because type-checking may not terminate. We introduced a lazy type-checking algorithm that will detect illegal cycles, which are are a common cause of non-termination.

Type-checking in DEEP-- is unusual because partial evaluation is built in to the type system. As described in Section 2.8, the algorithm for finding the principle or minimal supertype of a term involves reducing that term to normal form. It is a well known fact that reducing a term will often lead to a more specific type [Pierce, 2002]. The act of assigning a type to a term will thus partially evaluate the term; the "type" of 1 + 2 is 3. Typing and code generation go hand in hand.

The partial evaluator used by DEEP-- performs online evaluation [Jones et al., 1993]. The programmer must guide the partial evaluator with inline directives in order to ensure that evaluation terminates.

Because partial evaluation and type-checking are done at the same time, the partial evaluator can make use of typing information to perform reductions that would not be possible in an untyped calculus. In particular, it is able to do inlining of final bindings.

6.2.1 Mixins and multiple inheritance

Chapter 4 introduced the DEEP calculus, which resolves the most serious limitation of DEEP--. Unlike DEEP--, it is possible in DEEP to inherit from a virtual class or from a variable. Not only does this mean that DEEP supports higher order hierarchies, but it supports a form of multiple inheritance based on mixin composition.

We showed that mixins can be encoded as functions over classes (or modules)

which are monotonic with respect to subtyping. Monotonicity is enforced using Steffen's *polarized higher-order subtyping* [Steffen, 1997]. The familiar semantics of multiple inheritance emerge naturally from this encoding (see Section 4.3.5).

Although polarized higher-order subtyping was developed a decade ago, we have never seen it used in this way before. Multiple inheritance is usually modeled with intersection types [Compagnoni and Pierce, 1996]. We showed that intersection types are not only unnecessary, they are actually inferior where mixins are concerned. Our encoding of mixins takes the ordering of a mixin composition into account. Although different orderings have the same interface, they yield different behaviors in practice, and we believe that this fact should be reflected in the type theory.

We showed that the DEEP calculus is strong enough to solve the expression problem in two different ways. It supports solutions in both the object-oriented and functional programming styles, which were described in Chapter 1. We also gave a solution to the tag elimination problem, which makes use of the built-in partial evaluator. Finally, we gave a solution to the DSL expression problem, which combines the expression problem and the tag elimination problem into a single example. Our solution has been tested with the current implementation of DEEP.

A significant limitation of DEEP is the fact that we were only able to solve the DSL expression problem using an object-oriented programming style. A functional solution requires more sophisticated pattern matching than we were able to easily encode.

6.3 Discussion of practical experience

Our experience with using DEEP as a language for practical programming has been quite positive thus far. We have not done any large-scale studies, because the current implementation of DEEP is currently at the prototype stage, and is not very robust at this time. Nevertheless, our experience has convinced us that delegation is a powerful and intuitive mechanism for specifying inheritance between both classes and modules. The uniform treatment of types and objects is also convenient, because any valid expression can be used as a type. This means that types can be stored in data structures, or computed by recursive functions.

In practice, decidability does not seem to be an issue. Performing recursive computations with types is similar to performing such computations with objects; the same amount of care is required in both cases. We have not observed any cases where the type-checker hangs unexpectedly, although this could be because we are deeply

familiar (pun intended) with the way type checking is implemented. We do note that accidental non-termination is unlikely because subtyping between modules is nominal; the type system never does a structural comparison of recursive modules.

Perhaps most importantly, when porting a Java or a Haskell program to DEEP, where the program only contains simple types and basic polymorphism, the type system behaves the same way it does in Java or Haskell; the extra expressive power does not come into play. The only real difficulty that we have encountered when working with DEEP is that the expressiveness of the type system makes certain things look possible that actually aren't possible, or at least are very difficult; pattern matching for GADTs is a good example.

Once again, these observations are based on experimentation with small programs, rather than large pieces of software which have been deployed in the real world. A great deal more practical experience is required before we can come to any definitive conclusions.

6.4 Discussion of theoretical results

The most obvious flaw in the design of DEEP is not practical, but theoretical: we have not been able to prove type safety. The meta-theoretic difficulties that we have encountered raise an obvious question. Would adding types to the language improve matters? Is it possible to implement virtual classes and recursive mixin modules within a more traditional type theory, which would be provably type-safe, and perhaps even have decidable typing? This is a very difficult question to answer, but it is one that we have thought about at some length.

As explained in Chapter 2, the proof of transitivity elimination is difficult because System $\lambda_{\lhd}$ supports both higher-order subtyping and bounded quantification on type operators. As explained in Chapter 4, higher-order subtyping with bounded quantification provides an elegant model of multiple inheritance, and multiple inheritance of some kind is a requirement for mixin modules.

The only existing proof of transitivity elimination (for System $F^{\omega}_{\leq}$) requires the language of types to be strongly normalizing. Thus, adding types to the language will only help if we can separate types and objects sufficiently well that type expressions are guaranteed to terminate. This would certainly be a desirable state of affairs, since subtyping would also be decidable, and our typing-checking algorithms would be complete.

There are two issues that cause strong normalization to fail in our current theory. The first is Girard's paradox, and the second is the fact that modules in DEEP-- and DEEP are recursive. Both of these issues are difficult to avoid.

We could avoid Girard's paradox in System $\lambda_{\lhd}$ by eliminating Top. However, Girard's paradox can also rear its head for other reasons, such as the presence of impredicative strong existential sums [Hook and Howe, 1986]. Modules are strong existential sums, because both type and object members can be projected from a module. If a module is treated as an object that contains type members, then modules would also be impredicative; such sums are not allowed in dependently-typed calculi like ECC [Luo, 1994]. Even without fixpoints, it is therefore not possible to have both first-class modules and dependent types in a strongly normalizing theory.

Recursion is also hard to avoid. The syntax of a recursive module is specified using a single fixpoint. In order to properly quarantine recursion within the object level, this fixpoint must be split into a fixpoint over objects, and a fixpoint over types [Crary et al., 1999]. Such phase- splitting is not necessarily possible in the presence of dependent types. Morever, one must still ensure that the the resulting recursive types are contractive, which is not easy to do in the presence of late binding.

Realistically, the best way to ensure that recursive type definitions are contractive is to abandon the higher-order types of System $F^{\omega}_{\leq}$, and instead restrict parameterized types to first-order definitions, as is done in Java 1.5, most virtual class calculi, and Duggan and Sourelis' mixin modules [Duggan and Sourelis, 1996].

A first-order type system allows datatypes and class definitions to be parameterized, such as List<T>, but does not allow List to be used as a naked function in which T is unspecified. A higher-order system supports arbitrary functions over types, and allows such functions to be passed as parameters to other types. Since our theory of mixins is based on higher-order types, we feel that it would be a mistake to omit them.

In summary, the presence of first-class recursive mixin modules makes it difficult to guarantee strong normalization of types without abandoning both dependent types, and higher-order types. We were unwilling to make that sacrifice. If the language of types is not strongly normalizing, then there is little point in trying to distinguish between types and objects; such a distinction would add complexity without solving any of our meta-theoretic difficulties.

6.5 Conclusion

The problem that we set out to solve in this thesis is not an easy one. We wanted to design a language that was truly extensible, one that would allow us to seamlessly embed other domain-specific languages inside of it.

In order for an embedding to be "seamless", it must have several characteristics. First, the embedding should be efficient: there should be no difference in execution speed between code written in the domain-specific language, and code written directly in the host programming language. Second, the embedding should be type-safe: the type-checking phase in the host language should catch type errors in the domain-specific language. Third, the domain-specific language itself should be extensible with new constructs.

The "DSL expression problem" codifies these requirements into a precise problem description. A solution to the DSL expression problem is an interpreter for a domain-specific language that is extensible, type-safe, and efficient.

The goal of efficiency requires code generation of some kind. We chose partial evaluation because it can guarantee type safety. Unlike most other generative programming techniques, partial evaluators perform type-checking before code generation, rather than after code generation, which makes type errors much easier to decipher. Moreover, research on partial evaluation has long focused on automatically transforming intepreters into compilers, which makes it ideal for implementing domain-specific languages.

However, partial evaluation on its own is not a silver bullet. The DSL expression problem consists of two difficult sub-problems: the expression problem, and the tag-elimination problem. Both of these sub-problems require a sophisticated type system. We needed a type theory that could handle first-class recursive mixin modules, with virtual classes, subtyping, and dependent types. This combination of features is not provided by any existing type theory in the litature.

This thesis presents a type theory that is capable of handling all these things, and we have used that theory to devise a solution to the DSL expression problem, which has never been done before.

Although we succeeded in our goal of solving the DSL expression problem, there is a great deal more work to be done. On the theoretical side, a complete proof of type safety is obviously the most pressing concern. Without such a proof, it is hard to claim that the DSL expression problem has been truly "solved". Some ideas for future

research in this area are presented in Section 2.9.

On the practical side, we would like to improve the DEEP linking mechanism, and add support for dynamic types and GADT-style pattern matching, as described in Section 4.9. A more robust implementation of DEEP, combined with further large-scale studies, are also necessary in order to demonstrate that the language is suitable for real-world software engineering.

In summary, we have developed a new and interesting approach to type theory, and we have used that approach to solve a problem of immediate practical interest. Our solution lays the groundwork for further research in this area.

Appendix A

Summary of formal systems

Syntax:

x, y, z	Variable		
$s, t, u ::=$	Terms	$\Gamma ::=$	Contexts
x	variable	$\emptyset$	empty context
Top	universal supertype	$\Gamma, x \leq t$	variable
$\lambda x \leq t.\ u$	function		
$t(u)$	application	$\lhd ::=$	Type relations
		$\leq$	subtype
$v, w ::=$	Values	$\equiv$	type equivalence
Top	universal supertype		
$\lambda x \leq t.\ u$	function		

Context:
$C ::= [] \mid C(t) \mid t(C) \mid \lambda x \leq C.\ t \mid \lambda x \leq t.\ C$

Operational Semantics (Reduction): $\boxed{t \longrightarrow t'}$

$$\frac{t \longrightarrow t'}{C[t] \longrightarrow C[t']} \qquad \text{(E-Cong)}$$

$$(\lambda x \leq t.\ u)(s) \longrightarrow [x \mapsto s]u \quad \text{(E-App)}$$

Notation:

- $C[t]$ represents a context C with the term t substituted for the hole $[]$ in the context.
- $[x \mapsto t]u$ denotes the capture-avoiding substitution of the term t for the variable x within u.

Figure A.1: System $\lambda_{\lhd}$— syntax and operational semantics

Subtyping: $\boxed{\Gamma \vdash t \lhd u}$

$$\frac{\Gamma \vdash s \lhd t, \quad t \lhd u, \quad t \text{ wf}}{\Gamma \vdash s \lhd u} \text{ (DS-TRANS)}$$

$$\frac{\Gamma \vdash u \equiv t}{\Gamma \vdash t \equiv u} \text{ (DS-SYM)} \qquad \frac{\Gamma \vdash t \equiv u}{\Gamma \vdash t \leq u} \text{ (DS-EQ)}$$

$$\Gamma \vdash x \equiv x \text{ (DS-VAR)}$$

$$\Gamma \vdash \mathsf{Top} \equiv \mathsf{Top} \text{ (DS-TOP)}$$

$$\frac{\Gamma \vdash t \equiv t' \quad \Gamma, x \leq t \vdash u \lhd u'}{\Gamma \vdash \lambda x \leq t.\, u \lhd \lambda x \leq t'.\, u'} \text{ (DS-FUN)}$$

$$\frac{\Gamma \vdash t \lhd t', \quad u \equiv u'}{\Gamma \vdash t(u) \lhd t'(u')} \text{ (DS-APP)}$$

$$\Gamma \vdash t \leq \mathsf{Top} \text{ (DS-ETOP)}$$

$$\frac{x \leq t \in \Gamma}{\Gamma \vdash x \leq t} \text{ (DS-EVAR)}$$

$$\Gamma \vdash (\lambda x \leq t.\, u)(s) \equiv [x \mapsto s]u \text{ (DS-EAPP)}$$

Well-subtyping: $\boxed{\Gamma \vdash t \lhd_{\text{wf}} u}$

$$\frac{\Gamma \vdash t \text{ wf}, \quad u \text{ wf}, \quad t \lhd u}{\Gamma \vdash t \lhd_{\text{wf}} u} \text{ (W-SUB)}$$

Context well-formedness: $\boxed{\Gamma \text{ wf}}$

$$\emptyset \text{ wf} \text{ (W-GAM1)}$$

$$\frac{\Gamma \text{ wf} \quad x \notin \text{dom}(\Gamma) \quad \Gamma \vdash t \text{ wf}}{\Gamma, x \leq t \text{ wf}} \text{ (W-GAM2)}$$

Term well-formedness: $\boxed{t \text{ wf}}$

$$\frac{\Gamma \text{ wf} \quad x \in \text{dom}(\Gamma)}{\Gamma \vdash x \text{ wf}} \text{ (W-VAR)}$$

$$\frac{\Gamma \text{ wf}}{\Gamma \vdash \mathsf{Top} \text{ wf}} \text{ (W-TOP)}$$

$$\frac{\Gamma, x \leq t \vdash u \text{ wf}}{\Gamma \vdash \lambda x \leq t.\, u \text{ wf}} \text{ (W-FUN)}$$

$$\frac{\Gamma \vdash t \leq_{\text{wf}} \lambda x \leq s.\, \mathsf{Top} \quad \Gamma \vdash u \leq_{\text{wf}} s}{\Gamma \vdash t(u) \text{ wf}} \text{ (W-APP)}$$

Notation:

- $\text{fv}(t)$ denotes the set of free variables in the term t.
- $\text{dom}(\Gamma)$ denotes the set of variables defined in Γ.
- $x \leq t \in \Gamma$ is true if the context Γ contains $x \leq t$.

We identify terms which are α-equivalent under renaming of bound variables. As usual, we assume that variables can be renamed as needed to avoid conflicts.
For compactness, we adopt the following convention: a pair of judgements $\Gamma \vdash J_1$ and $\Gamma \vdash J_2$ which are both made within the same context Γ are written as $\Gamma \vdash J_1, J_2$.

Note that $\lhd$ is a meta-variable which ranges over $\leq$ and $\equiv$.

Figure A.2: System $\lambda_{\lhd}$— declarative subtyping and well-formedness

Prevalidity: $\boxed{\Gamma \text{ prevalid}}$

$$\emptyset \text{ prevalid} \qquad \text{(P-CTX1)}$$

$$\frac{\Gamma \text{ prevalid} \quad x \notin \text{dom}(\Gamma) \quad \text{fv}(t) \subseteq \text{dom}(\Gamma)}{\Gamma, x \leq t \text{ prevalid}} \qquad \text{(P-CTX2)}$$

Subtyping: $\boxed{\Gamma \vdash_A t \lhd u}$

$$\frac{\Gamma \text{ prevalid}}{\Gamma \vdash_A t \lhd t} \qquad \text{(AS-REFL)}$$

$$\frac{\Gamma \vdash_A t \xrightarrow{\lhd} t', \quad t' \lhd u}{\Gamma \vdash_A t \lhd u} \qquad \text{(AS-LEFT)}$$

$$\frac{\Gamma \vdash_A u \xrightarrow{\equiv} u', \quad t \lhd u'}{\Gamma \vdash_A t \lhd u} \qquad \text{(AS-RIGHT)}$$

Transitive Subtyping: $\boxed{\Gamma \vdash_A t \lhd^* u}$

$$\frac{\Gamma \vdash_A s \lhd u}{\Gamma \vdash_A s \lhd^* u} \qquad \text{(AST-SUB)}$$

$$\frac{\Gamma \vdash_A s \lhd^* t, \quad t \lhd^* u}{\Gamma \vdash_A s \lhd^* u} \qquad \text{(AST-TRANS)}$$

Subtype reduction: $\boxed{\Gamma \vdash_A t \xrightarrow{\leq} t'}$

$$\frac{\Gamma \text{ prevalid} \quad x \leq t \in \Gamma}{\Gamma \vdash_A x \xrightarrow{\leq} t} \qquad \text{(SRS-PROM)}$$

$$\frac{\Gamma \text{ prevalid}}{\Gamma \vdash_A t \xrightarrow{\leq} \mathsf{Top}} \qquad \text{(SRS-TOP)}$$

Equivalence reduction: $\boxed{\Gamma \vdash_A t \xrightarrow{\equiv} t'}$

$$\frac{\Gamma \vdash_A s \leq^* t}{\Gamma \vdash_A (\lambda x \leq t.\, u)(s) \xrightarrow{\equiv} [x \mapsto s]u} \qquad \text{(SRE-APP)}$$

$$\frac{\Gamma \text{ prevalid}}{\Gamma \vdash_A \mathsf{Top}(t) \xrightarrow{\equiv} \mathsf{Top}} \qquad \text{(SRE-TOPAPP)}$$

Congruence rules: $\boxed{\Gamma \vdash_A t \xrightarrow{\lhd} t'}$

$$E_{\equiv} ::= [] \mid E_{\equiv}(t) \mid t(E_{\equiv}) \mid \lambda x \leq E_{\equiv}.\, t$$

$$E_{\leq} ::= [] \mid E_{\leq}(t)$$

$$\frac{\Gamma \vdash_A t \xrightarrow{\lhd} t'}{\Gamma \vdash_A E_{\lhd}[t] \xrightarrow{\lhd} E_{\lhd}[t']} \qquad \text{(SR-CONG)}$$

$$\frac{\Gamma, x \leq t \vdash_A u \xrightarrow{\lhd} u'}{\Gamma \vdash_A \lambda x \leq t.\, u \xrightarrow{\lhd} \lambda x \leq t.\, u'} \qquad \text{(SR-FUN)}$$

$$\frac{\Gamma \vdash_A t \xrightarrow{\equiv} t'}{\Gamma \vdash_A t \xrightarrow{\leq} t'} \qquad \text{(SR-EQ)}$$

Figure A.3: System $\lambda_{\lhd}$— algorithmic subtyping

x, y, z	variable	O ::= def | override	modifier
ℓ, l, m	slot name	$\doteq$ ::= : | $=$	virtual/final
s, t, u ::=	terms	v, w ::=	values
x	variable	Top	Top-type
Top	Top-type	$\lambda x \leq t.\ u$	function
$\lambda x \leq t.\ u$	function	μx extends t $\{\overline{l}\ \overline{u}\}$	module
μx extends t $\{\overline{d}\}$	module	$: t$ inline$(\overline{s}) = u$	field
$: t$ inline$(\overline{s}) = u$	field	Γ ::=	contexts
$t(u)$	apply	$\emptyset$	empty context
$t@(u).l$	delegate	$\Gamma, x \leq t$	upper bound
$t\$$	extract	$\lhd$ ::=	type relations
d, e, f ::=	declarations	$\leq$	subtype
$O\ l \doteq t$	labeled term	$\equiv$	equivalence

Notation:

- $\overline{t}$ denotes a sequence of zero or more terms, separated by commas.
- $\overline{d}$ denotes a sequence of zero or more declarations, separated by semicolons.
- dom$(\overline{d})$ denotes the set of labels in the sequence of declarations $\overline{d}$.
- $e \in \overline{d}$ is true if e appears in the sequence $\overline{d}$.

Equality: Terms are considered to be syntactically equal (i.e. α-equivalent) under renaming of bound variables, plus the following rule:

$$\frac{\overline{d} \text{ is a permutation of } \overline{e}}{\mu x \text{ extends } t\ \{\overline{d}\}\ =\ \mu x \text{ extends } t\ \{\overline{e}\}}$$

Evaluation context:

$C ::= [] \mid C(t) \mid t(C) \mid C@(t).l \mid t@(C).l \mid C\$ \mid \lambda x \leq C.\ t \mid \lambda x \leq t.\ C \mid \mu x \text{ extends } C\ \{\overline{d}\}$
$\mid \mu x \text{ extends } t\ \{\overline{d};\ O\ l \doteq C;\ \overline{e}\} \mid\ : C \text{ inline}(\overline{s}) = u \mid\ : t \text{ inline}(\overline{s}, C, \overline{s'}) = u \mid\ : t \text{ inline}(\overline{s}) = C$

$$\frac{t \longrightarrow t'}{C[t] \longrightarrow C[t']} \qquad \text{(E-Cong)}$$

$$(\lambda x \leq t.\ u)(s) \longrightarrow [x \mapsto s]u \qquad \text{(E-App)}$$

$$\frac{O\ l \doteq u \in \overline{d}}{(\mu x \text{ extends } t\ \{\overline{d}\})@(s).l \longrightarrow [x \mapsto s]u} \qquad \text{(E-Dlg1)}$$

$$\frac{l \notin \text{dom}(\overline{d})}{(\mu x \text{ extends } t\ \{\overline{d}\})@(s).l \longrightarrow t@(s).l} \qquad \text{(E-Dlg2)}$$

$$(: t \text{ inline}(\overline{s}) = u)\$ \quad \longrightarrow \quad u \qquad \text{(E-Ext)}$$

Figure A.4: The Deep-- calculus — surface syntax and operational semantics

x, y, z	variables	Program ::=	program
ℓ, l, m	slot names	let $\overline{G} = \overline{M}$ in t	letrec
G, H	global variables		
		M, N ::=	module definition
s, t, u ::=	terms	μx extends t $\{\overline{d}\}$	module
x	variable	$\lambda x \leq t.\ M$	parameterized
Top	Top-type		
$\lambda x \leq t.\ u$	function	d, e, f ::=	declarations
$G(\overline{t})$	module	$O\ l \doteqdot t$	labeled term
$: t$ inline$(\overline{s}) = u$	field		
$t(u)$	apply	O ::= def | override	modifier
$t@(u).l$	delegate	$\doteqdot$ ::= : | =	virtual/final
$t\$$	extract		
		Γ ::=	contexts
v, w ::=	values	$\emptyset$	empty context
Top	Top-type	$\Gamma, x \leq t$	upper bound
$\lambda x \leq t.\ u$	function		
$G(\overline{t})$	module	$\lhd$ ::=	type relations
$: t$ inline$(\overline{s}) = u$	field	$\leq$	subtype
		$\equiv$	equivalence

Notation:

- $\overline{t}$ denotes a sequence of zero or more terms, separated by commas.
- $\overline{d}$ denotes a sequence of zero or more declarations, separated by semicolons.
- dom$(\overline{d})$ denotes the set of labels in the sequence of declarations $\overline{d}$.
- $GT(G_i)$ denotes the module definition named G_i in the global table defined by the program: let $\overline{G} = \overline{M}$ in t.

Evaluation Context:

$$C ::= \ [] \mid C(t) \mid t(C) \mid C@(t).l \mid t@(C).l \mid C\$ \mid G(\overline{t}, C, \overline{u})$$
$$\mid \ \lambda x \leq C.\ t \mid \lambda x \leq t.\ C \mid \ : C \text{ inline}(\overline{s}) = u \mid \ : t \text{ inline}(\overline{s}, C, \overline{s'}) = u \mid \ : t \text{ inline}(\overline{s}) = C$$

Reduction: $\boxed{t \longrightarrow t'}$

$$\frac{t \longrightarrow t'}{C[t] \longrightarrow C[t']} \quad \text{(E-Cong)}$$

$$\frac{G(\overline{t}) \rightsquigarrow \mu x \text{ extends } t' \ \{\overline{d}\} \quad O\ l \doteqdot u \in \overline{d}}{G(\overline{t})@(s).l \longrightarrow [x \mapsto s]u} \quad \text{(E-Dlg1)}$$

$$\frac{G(\overline{t}) \rightsquigarrow \mu x \text{ extends } t' \ \{\overline{d}\} \quad l \notin \text{dom}(\overline{d})}{G(\overline{t})@(s).l \longrightarrow t'@(s).l} \quad \text{(E-Dlg2)}$$

$$(\lambda x \leq t.\ u)(s) \longrightarrow [x \mapsto s]u \quad \text{(E-App)}$$

$$(: t \text{ inline}(\overline{s}) = u)\$ \longrightarrow u \quad \text{(E-Ext)}$$

Global module lookup: $\boxed{G(\overline{t}) \rightsquigarrow M}$

$$G() \rightsquigarrow GT(G) \quad \text{(E-Lookup1)}$$

$$\frac{G(\overline{t}) \rightsquigarrow \lambda x \leq s.\ M}{G(\overline{t}, u) \rightsquigarrow [x \mapsto u]M} \quad \text{(E-Lookup2)}$$

Figure A.5: The Deep-- calculus — syntax and operational semantics

Well-subtyping: $\boxed{\Gamma \vdash\ t \leq_{\text{wf}} u}$

$$\frac{\Gamma \vdash\ t\ \text{wf}, \quad u\ \text{wf}, \quad t \lhd u}{\Gamma \vdash\ t \lhd_{\text{wf}} u} \quad \text{(W-SUB)}$$

Subtyping & equivalence: $\boxed{\Gamma \vdash\ t \lhd u}$

$$\frac{\Gamma \vdash\ s \lhd t, \quad t \lhd u, \quad t\ \text{wf}}{\Gamma \vdash\ s \lhd u} \text{(DS-TRANS)}$$

$$\frac{\Gamma \vdash\ u \equiv t}{\Gamma \vdash\ t \equiv u} \quad \text{(DS-SYM)} \qquad \frac{\Gamma \vdash\ t \equiv u}{\Gamma \vdash\ t \leq u} \quad \text{(DS-EQ)}$$

$$\frac{\begin{array}{c}\Gamma \vdash\ t \equiv t' \\ \Gamma, x \leq t \vdash\ u \lhd u'\end{array}}{\Gamma \vdash\ \lambda x \leq t.\ u \ \lhd\ \lambda x \leq t'.\ u'} \text{(DS-FUN)}$$

$$\frac{G() \rightsquigarrow M}{\Gamma \vdash\ G() \equiv G()} \quad \text{(DS-MOD1)}$$

$$\frac{\Gamma \vdash\ G(\overline{t}) \lhd G(\overline{t'}) \quad u \equiv u'}{\Gamma \vdash\ G(\overline{t}, u) \lhd G(\overline{t'}, u')} \text{(DS-MOD2)}$$

$$\Gamma \vdash\ \text{Top} \equiv \text{Top} \quad \text{(DS-TOP)}$$

$$\frac{x \in \text{dom}(\Gamma)}{\Gamma \vdash\ x \equiv x} \quad \text{(DS-VAR)}$$

$$\frac{\Gamma \vdash\ t \lhd t', \quad u \equiv u'}{\Gamma \vdash\ t(u) \lhd t'(u')} \quad \text{(DS-APP)}$$

$$\frac{\Gamma \vdash\ t \lhd t', \quad u \equiv u'}{\Gamma \vdash\ t@(u).l \lhd t'@(u').l} \quad \text{(DS-DLG)}$$

$$\frac{\Gamma \vdash\ t \equiv u}{\Gamma \vdash\ t\$ \equiv u\$} \quad \text{(DS-EXT)}$$

Equivalence (reduction): $\boxed{\Gamma \vdash\ t \equiv u}$

$$\Gamma \vdash (\lambda x \leq t.\ u)(s) \equiv [x \mapsto s]u \quad \text{(DS-EAPP)}$$

$$\frac{\begin{array}{c}G(\overline{t}) \rightsquigarrow \mu x\ \text{extends}\ t'\ \{\overline{d}\} \\ O\ l \doteqdot u \in \overline{d}\end{array}}{\Gamma \vdash\ G(\overline{t})@(s).l \equiv [x \mapsto s]u} \text{(DS-EDLG1)}$$

$$\frac{\begin{array}{c}G(\overline{t}) \rightsquigarrow \mu x\ \text{extends}\ t'\ \{\overline{d}\} \\ l \notin \text{dom}(\overline{d})\end{array}}{\Gamma \vdash\ G(\overline{t})@(s).l \equiv t'@(s).l} \text{(DS-EDLG2)}$$

$$\frac{\overline{v}\ \text{not empty}}{\Gamma \vdash\ (:t\ \text{inline}(\overline{v}) = u)\$ \equiv u} \quad \text{(DS-EEXT1)}$$

$$\frac{\begin{array}{c}\Gamma \vdash\ t \leq_{\text{wf}} G(\overline{t'}) \\ G(\overline{t'}) \rightsquigarrow \mu x\ \text{extends}\ s'\ \{\overline{d}\} \\ O\ l = u \in \overline{d}\end{array}}{\Gamma \vdash\ t@(s).l \equiv [x \mapsto s]u} \text{(DS-EFINAL)}$$

Subtyping (reduction): $\boxed{\Gamma \vdash\ t \leq u}$

$$\Gamma \vdash\ t \leq \text{Top} \quad \text{(DS-ETOP)}$$

$$\frac{x \leq t \in \Gamma}{\Gamma \vdash\ x \leq t} \quad \text{(DS-EVAR)}$$

$$\frac{G(\overline{t}) \rightsquigarrow \mu x\ \text{extends}\ u\ \{\overline{d}\}}{\Gamma \vdash\ G(\overline{t}) \leq u} \quad \text{(DS-EINH)}$$

$$\frac{\Gamma \vdash\ t \leq_{\text{wf}} (:u\ \text{inline}(\overline{s}) = u')}{\Gamma \vdash\ t\$ \leq u} \text{(DS-EEXT2)}$$

$$\frac{\Gamma \vdash\ t \equiv t'}{\Gamma \vdash\ (:t\ \text{inline}(\overline{s}) = u) \leq (:t'\ \text{inline}(\overline{s'}) = u')} \text{(DS-FIELD1)}$$

$$\frac{\Gamma \vdash\ t \equiv t', \quad u \equiv u', \quad \overline{s} \equiv \overline{s'}}{\Gamma \vdash\ (:t\ \text{inline}(\overline{s}) = u) \equiv (:t'\ \text{inline}(\overline{s'}) = u')} \text{(DS-FIELD2)}$$

Figure A.6: The DEEP-- calculus — declarative subtyping

Context well-formedness: $\boxed{\Gamma \text{ wf}}$

$$\emptyset \text{ wf} \qquad \text{(W-GAM1)}$$

$$\frac{\Gamma \text{ wf}, \quad x \notin \text{dom}(\Gamma) \quad \Gamma \vdash t \text{ wf}}{\Gamma, x \le t \text{ wf}} \qquad \text{(W-GAM2)}$$

Term well-formedness: $\boxed{t \text{ wf}}$

$$\frac{\Gamma \text{ wf} \quad x \in \text{dom}(\Gamma)}{\Gamma \vdash x \text{ wf}} \qquad \text{(W-VAR)}$$

$$\frac{\Gamma \text{ wf}}{\Gamma \vdash \mathsf{Top} \text{ wf}} \qquad \text{(W-TOP)}$$

$$\frac{\Gamma, x \le t \vdash u \text{ wf}}{\Gamma \vdash \lambda x \le t.\, u \text{ wf}} \qquad \text{(W-FUN)}$$

$$\frac{G() \rightsquigarrow M}{\Gamma \vdash G() \text{ wf}} \qquad \text{(W-MOD1)}$$

$$\frac{\Gamma \vdash G(\overline{t}) \text{ wf} \quad G(\overline{t}) \rightsquigarrow \lambda x \le s.\, M \quad \Gamma \vdash u \le_{\text{wf}} s}{\Gamma \vdash G(\overline{t}, u) \text{ wf}} \qquad \text{(W-MOD2)}$$

$$\frac{\Gamma \vdash \overline{s} \text{ wf}, \quad u \le_{\text{wf}} t}{\Gamma \vdash (: t\ \mathsf{inline}(\overline{s}) = u) \text{ wf}} \qquad \text{(W-FIELD)}$$

$$\frac{\Gamma \vdash t \le_{\text{wf}} (\lambda x \le s.\, \mathsf{Top}) \quad \Gamma \vdash u \le_{\text{wf}} s}{\Gamma \vdash t(u) \text{ wf}} \qquad \text{(W-APP)}$$

$$\frac{\Gamma \vdash t \le_{\text{wf}} G(\overline{s}) \quad G(\overline{s}) \rightsquigarrow \mu x\ \mathsf{extends}\ t'\ \{\overline{d}\} \quad O\ l \doteq u' \in \overline{d} \quad \Gamma \vdash u \le_{\text{wf}} t}{\Gamma \vdash t@(u).l \text{ wf}} \qquad \text{(W-DLG)}$$

$$\frac{\Gamma \vdash t \le_{\text{wf}} (: u\ \mathsf{inline}(\overline{s}) = u')}{\Gamma \vdash t\$ \text{ wf}} \qquad \text{(W-EXT)}$$

Program well-formedness:

$$\frac{\overline{G} :: \emptyset \vdash \overline{M} \text{ wf} \quad \emptyset \vdash t \text{ wf} \quad \text{There are no illegal cycles.}}{\mathsf{let}\ \overline{G} = \overline{M}\ \mathsf{in}\ t \text{ wf}} \qquad \text{(W-PROG)}$$

Module wf: $\boxed{G(\overline{s}) :: \Gamma \vdash M \text{ wf}}$

$$\frac{G(\overline{s}, x) :: \Gamma, x \le t \vdash M \text{ wf}}{G(\overline{s}) :: \Gamma \vdash \lambda x \le t.\, M \text{ wf}} \qquad \text{(W-MFUN)}$$

$$\frac{\Gamma \vdash t \text{ wf} \quad \Gamma, x \le G(\overline{s}) \vdash \overline{d} \text{ wf} \quad \overline{d} \text{ has no duplicate labels.}}{G(\overline{s}) :: \Gamma \vdash \mu x\ \mathsf{extends}\ t\ \{\overline{d}\} \text{ wf}} \qquad \text{(W-MDEF)}$$

Declaration well-formedness: $\boxed{\Gamma \vdash d \text{ wf}}$

$$\frac{\Gamma, x \le G(\overline{s}) \vdash t \text{ wf} \quad G(\overline{s}) \rightsquigarrow \mu y\ \mathsf{extends}\ u\ \{\overline{d}\} \quad \Gamma \vdash u.l \text{ undefined}}{\Gamma, x \le G(\overline{s}) \vdash \mathsf{def}\ l \doteq t \text{ wf}} \qquad \text{(W-DECL)}$$

$$\frac{\Gamma, x \le G(\overline{s}) \vdash t \le_{\text{wf}} u@(x).l \quad G(\overline{s}) \rightsquigarrow \mu y\ \mathsf{extends}\ u\ \{\overline{d}\} \quad \Gamma \vdash u.l \text{ virtual}}{\Gamma, x \le G(\overline{s}) \vdash \mathsf{override}\ l \doteq t \text{ wf}} \qquad \text{(W-ODECL)}$$

virtual/undefined:

$$\Gamma \vdash \mathsf{Top}.l \text{ virtual} \qquad \Gamma \vdash \mathsf{Top}.l \text{ undefined}$$

$$\frac{\Gamma \vdash t \equiv G(\overline{s}) \quad G(\overline{s}) \rightsquigarrow \mu x\ \mathsf{extends}\ t'\ \{\overline{d}\} \quad O\ l = u \notin \overline{d} \quad \Gamma \vdash t'.l \text{ virtual}}{\Gamma \vdash t.l \text{ virtual}}$$

$$\frac{\Gamma \vdash t \equiv G(\overline{s}) \quad G(\overline{s}) \rightsquigarrow \mu x\ \mathsf{extends}\ t'\ \{\overline{d}\} \quad l \notin \text{dom}(\overline{d}) \quad \Gamma \vdash t'.l \text{ undefined}}{\Gamma \vdash t.l \text{ undefined}}$$

Figure A.7: The DEEP-- calculus — well-formedness

x,y,z	variables	$d,e,f ::=$	declarations
ℓ,l,m	slot names	$\quad O\ l^* \doteqdot t$	labeled term
G,H	global variables	$O ::= \mathsf{def} \mid \mathsf{override}$	modifier
		$* ::= \pm \mid +$	polarity
$\mathsf{Program} ::=$	program	$\doteqdot ::= : \mid =$	virtual/final
$\quad \mathsf{let}\ \overline{G} = \overline{M}\ \mathsf{in}\ t$	letrec	$\sigma ::= \surd \mid ?$	linked/unlinked
$M,N ::=$	module definition	$v,w ::=$	values
$\quad \mu x\ \mathsf{extends}\ t\ \{\overline{d}\}$	module	$\quad \mathsf{Top}$	Top-type
$\quad \lambda^* x \le t.\ M$	param. module	$\quad \lambda^*_\sigma x \le t.\ u$	function
		$\quad G(\bar{t})_\surd$	linked module
$s,t,u ::=$	terms	$\quad : t\ \mathsf{inline}(\bar{s}) = u$	field
$\quad x$	variable		
$\quad \mathsf{Top}$	Top-type	$\Gamma ::=$	contexts
$\quad \lambda^*_\sigma x \le t.\ u$	function	$\quad \emptyset$	empty context
$\quad G(\bar{t})_\sigma$	module	$\quad \Gamma, x \le t$	upper bound
$\quad : t\ \mathsf{inline}(\bar{s}) = u$	field		
$\quad t(u)^*_\sigma$	apply	$\lhd ::=$	type relation
$\quad t@(u).l^*$	delegate	$\quad \le$	subtype
$\quad t\$$	extract	$\quad \cong$	congruence
		$\quad \equiv$	equivalence

Notation:

- $\bar{t}$ and $\overline{d}$ denote a sequence of zero or more terms or declarations, respectively.
- $\mathrm{dom}(\overline{d})$ denotes the set of labels in the sequence of declarations $\overline{d}$.
- $GT(G_i)$ denotes the module named G_i in the program: let $\overline{G} = \overline{M}$ in t.

Evaluation Context:

$$C ::= [] \mid C(t)^*_\sigma \mid t(C)^*_\sigma \mid C@(t).l^* \mid t@(C).l^* \mid C\$ \mid G(\bar{t}, C, \bar{u})_\sigma$$
$$\mid \lambda^*_\sigma x \le C.\ t \mid \lambda^*_\sigma x \le t.\ C \mid : C\ \mathsf{inline}(\bar{s}) = u \mid : t\ \mathsf{inline}(\bar{s}, C, \bar{s'}) = u \mid : t\ \mathsf{inline}(\bar{s}) = C$$

Reduction: $\boxed{t \longrightarrow t'}$

$$(\lambda^*_\sigma x \le t.\ u)(s)^*_\sigma \longrightarrow [x \mapsto s]u \quad \text{(E-App)}$$

$$\frac{G(\bar{t}) \rightsquigarrow \mu x\ \mathsf{extends}\ t'\ \{\overline{d}\} \quad O\ l^* \doteqdot u \in \overline{d}}{G(\bar{t})_\surd @(s).l^* \longrightarrow [x \mapsto s]u} \quad \text{(E-Dlg1)}$$

$$\frac{G(\bar{t}) \rightsquigarrow \mu x\ \mathsf{extends}\ t'\ \{\overline{d}\} \quad l \notin \mathrm{dom}(\overline{d})}{G(\bar{t})_\surd @(s).l^* \longrightarrow t'@(s).l^*} \quad \text{(E-Dlg2)}$$

$$(: t\ \mathsf{inline}(\bar{s}) = u)\$ \longrightarrow u \quad \text{(E-Ext)}$$

$$G(\bar{s})_? \longrightarrow G(\bar{s})_\surd \quad \text{(E-Link)}$$

$$\frac{t \longrightarrow t'}{C[t] \longrightarrow C[t']} \quad \text{(E-Cong)}$$

Global module lookup: $\boxed{G(\bar{t}) \rightsquigarrow M}$

$$G() \rightsquigarrow GT(G) \quad \text{(E-Look1)}$$

$$\frac{G(\bar{t}) \rightsquigarrow \lambda^* x \le s.\ M}{G(\bar{t}, u) \rightsquigarrow [x \mapsto u]M} \quad \text{(E-Look2)}$$

Figure A.8: The DEEP calculus — syntax and operational semantics

Well-subtyping: $\boxed{\Gamma \vdash\ t \lhd_{\text{wf}} u}$

$$\frac{\Gamma \vdash\ t\ \text{wf}, \quad u\ \text{wf}, \quad t \lhd u}{\Gamma \vdash\ t \lhd_{\text{wf}} u} \quad \text{(W-Sub)}$$

Subtyping & equivalence: $\boxed{\Gamma \vdash\ t \lhd u}$

$$\frac{\Gamma \vdash\ s \lhd t, \quad t \lhd u, \quad t\ \text{wf}}{\Gamma \vdash\ s \lhd u} \quad \text{(DS-Trans)}$$

$$\frac{\Gamma \vdash\ u \equiv t}{\Gamma \vdash\ t \equiv u} \qquad \frac{\Gamma \vdash\ u \cong t}{\Gamma \vdash\ t \cong u} \quad \text{(DS-Sym1-2)}$$

$$\frac{\Gamma \vdash\ t \equiv u}{\Gamma \vdash\ t \cong u} \quad \text{(DS-Cong)} \qquad \frac{\Gamma \vdash\ t \cong u}{\Gamma \vdash\ t \leq u} \quad \text{(DS-Eq)}$$

$$\frac{\begin{array}{c}\Gamma \vdash\ t \equiv t' \\ \Gamma, x \leq t \vdash\ u \lhd u'\end{array}}{\Gamma \vdash\ \lambda^*_\sigma x \leq t.\ u \ \lhd\ \lambda^*_\sigma x \leq t'.\ u'} \quad \text{(DS-Fun)}$$

$$\frac{G() \rightsquigarrow M}{\Gamma \vdash\ G()_\sigma \equiv G()_\sigma} \quad \text{(DS-Mod1)}$$

$$\frac{\Gamma \vdash\ G(\bar{t})_\sigma \lhd G(\overline{t'})_\sigma \quad u \equiv u'}{\Gamma \vdash\ G(\bar{t}, u)_\sigma \lhd G(\overline{t'}, u')_\sigma} \quad \text{(DS-Mod2)}$$

$$\frac{\begin{array}{c}G(\bar{t}) \rightsquigarrow \lambda^+ x \leq s.\ M \\ \Gamma \vdash\ G(\bar{t})_\sigma \leq G(\overline{t'})_\sigma, \quad u \leq u'\end{array}}{\Gamma \vdash\ G(\bar{t}, u)_\sigma \leq G(\overline{t'}, u')_\sigma} \quad \text{(DS-Mod2+)}$$

$$\Gamma \vdash\ \text{Top} \equiv \text{Top} \quad \text{(DS-Top)}$$

$$\frac{x \in \text{dom}(\Gamma)}{\Gamma \vdash\ x \equiv x} \quad \text{(DS-Var)}$$

$$\frac{\Gamma \vdash\ t \lhd t', \quad u \equiv u'}{\Gamma \vdash\ t(u)^*_\sigma \lhd t'(u')^*_\sigma} \quad \text{(DS-App)}$$

$$\frac{\Gamma \vdash\ t \leq t', \quad u \leq u'}{\Gamma \vdash\ t(u)^+_\sigma \leq t'(u')^+_\sigma} \quad \text{(DS-App+)}$$

$$\frac{\Gamma \vdash\ t \lhd t', \quad u \equiv u'}{\Gamma \vdash\ t@(u).l^* \lhd t'@(u').l^*} \quad \text{(DS-Dlg)}$$

$$\frac{\Gamma \vdash\ t \leq t', \quad u \leq u'}{\Gamma \vdash\ t@(u).l^+ \leq t'@(u').l^+} \quad \text{(DS-Dlg+)}$$

$$\frac{\Gamma \vdash\ t \equiv u}{\Gamma \vdash\ t\$ \equiv u\$} \quad \text{(DS-Ext)}$$

$$\Gamma \vdash\ t \leq \text{Top} \quad \text{(DS-ETop)}$$

$$\frac{x \leq t \in \Gamma}{\Gamma \vdash\ x \leq t} \quad \text{(DS-EVar)}$$

$$\frac{G(\bar{t}) \rightsquigarrow \mu x\ \text{extends}\ u\ \{\bar{d}\}}{\Gamma \vdash\ G(\bar{t})_\sigma \leq u} \quad \text{(DS-EInh)}$$

$$\frac{\Gamma \vdash\ t \leq_{\text{wf}} (:u\ \text{inline}(\bar{s}) = u')}{\Gamma \vdash\ t\$ \leq u} \quad \text{(DS-EExt1)}$$

$$\frac{\Gamma \vdash\ t \equiv t'}{\Gamma \vdash\ (:t\ \text{inline}(\bar{s}) = u) \cong (:t'\ \text{inline}(\overline{s'}) = u')} \quad \text{(DS-Field1)}$$

$$\frac{\Gamma \vdash\ t \equiv t', \quad u \equiv u', \quad \bar{s} \equiv \overline{s'}}{\Gamma \vdash\ (:t\ \text{inline}(\bar{s}) = u) \equiv (:t'\ \text{inline}(\overline{s'}) = u')} \quad \text{(DS-Field2)}$$

Figure A.9: The Deep calculus — declarative subtyping

Equivalence (reduction): $\boxed{\Gamma \vdash\ t \equiv t'}$

$$\Gamma \vdash (\lambda^*_\sigma x \le t.\ u)(s^k)^*_\sigma \equiv [x \mapsto s^k]u \quad \text{(DS-EAPP)}$$

$$\frac{G(\overline{t^k}) \rightsquigarrow \mu x \text{ extends } t' \ \{\overline{d}\} \qquad O\ l^* \doteqdot u \in \overline{d}}{\Gamma \vdash\ G(\overline{t^k})_\surd @ (s^k).l^* \equiv [x \mapsto s^k]u} \quad \text{(DS-EDLG1)}$$

$$\frac{G(\overline{t^k}) \rightsquigarrow \mu x \text{ extends } t' \ \{\overline{d}\} \qquad l \notin \mathrm{dom}(\overline{d})}{\Gamma \vdash\ G(\overline{t^k})_\surd @ (s^k).l^* \equiv t' @ (s^k).l^*} \quad \text{(DS-EDLG2)}$$

$$\frac{\overline{v} \text{ not empty}}{\Gamma \vdash (:t\ \mathsf{inline}(\overline{v}) = u)\$ \equiv u} \quad \text{(DS-EEXT)}$$

$$\frac{\Gamma \vdash\ t^k \le_{\mathrm{wf}} G(\overline{t'})_\sigma \qquad G(\overline{t'}) \rightsquigarrow \mu x \text{ extends } s' \ \{\overline{d}\} \qquad O\ l^* = u \in \overline{d}}{\Gamma \vdash\ t^k @ (s^k).l^* \equiv [x \mapsto s^k]u} \quad \text{(DS-EFINAL)}$$

$$\frac{\Gamma \vdash\ G(\bar{t}) \text{ wlk}}{\Gamma \vdash\ G(\bar{t})_? \equiv G(\bar{t})_\surd} \quad \text{(DS-ELINK)}$$

$$\frac{G(\overline{t^k}) \rightsquigarrow \mu x \text{ extends } t' \ \{\overline{d}\} \qquad O\ l^* \doteqdot u \in \overline{d}}{\Gamma \vdash\ G(\overline{t^k})_? @ (s^k).l^* \le [x \mapsto s^k]u} \quad \text{(DS-EDLGSUB)}$$

Congruence (η): $\boxed{\Gamma \vdash\ t \cong t'}$

$$\frac{\Gamma \vdash\ t \le_{\mathrm{wf}} \lambda^*_\sigma x \le s.\ \mathsf{Top}}{\Gamma \vdash\ t \cong \lambda^*_\sigma x \le s.\ t(x)^*_\sigma} \quad \text{(DS-ηFUN)}$$

$$\frac{\Gamma \vdash\ t \le_{\mathrm{wf}} (:u\ \mathsf{inline}(\overline{s}) = u')}{\Gamma \vdash\ t \cong (:u\ \mathsf{inline}() = t\$)} \quad \text{(DS-ηFIELD)}$$

Figure A.10: The DEEP calculus — type equivalence and congruence

Context well-formedness: $\boxed{\Gamma \text{ wf}}$

$$\emptyset \text{ wf} \qquad \text{(W-GAM1)}$$

$$\frac{\Gamma \text{ wf}, \quad x \notin \text{dom}(\Gamma) \quad \Gamma \vdash t \text{ wf}}{\Gamma, x \leq t \text{ wf}} \qquad \text{(W-GAM2)}$$

Term well-formedness: $\boxed{t \text{ wf}}$

$$\frac{\Gamma \text{ wf} \quad x \in \text{dom}(\Gamma)}{\Gamma \vdash x \text{ wf}} \qquad \text{(W-VAR)}$$

$$\frac{\Gamma \text{ wf}}{\Gamma \vdash \mathsf{Top} \text{ wf}} \qquad \text{(W-TOP)}$$

$$\frac{\Gamma, x \leq t \vdash u \text{ wf} \quad x \notin \text{npvars}(*, u)}{\Gamma \vdash \lambda^*_\sigma x \leq t.\, u \text{ wf}} \qquad \text{(W-FUN)}$$

$$\frac{G() \rightsquigarrow M}{\Gamma \vdash G()_? \text{ wf}} \qquad \text{(W-MOD1)}$$

$$\frac{\Gamma \vdash G(\bar{t})_? \text{ wf} \quad G(\bar{t}) \rightsquigarrow (\lambda^* x \leq s.\, M) \quad \Gamma \vdash u \leq_{\text{wf}} s}{\Gamma \vdash G(\bar{t}, u)_? \text{ wf}} \qquad \text{(W-MOD2)}$$

$$\frac{\Gamma \vdash \bar{s} \text{ wf}, \quad u \leq_{\text{wf}} t}{\Gamma \vdash (: t \ \mathsf{inline}(\bar{s}) = u) \text{ wf}} \qquad \text{(W-FIELD)}$$

$$\frac{\Gamma \vdash t \leq_{\text{wf}} (\lambda^*_\sigma x \leq s.\, \mathsf{Top}) \quad \Gamma \vdash u \leq_{\text{wf}} s}{\Gamma \vdash t(u)^*_\sigma \text{ wf}} \qquad \text{(W-APP)}$$

$$\frac{\Gamma \vdash t \leq_{\text{wf}} G(\bar{s})_\sigma \quad G(\bar{s}) \rightsquigarrow \mu x \ \mathsf{extends}\ t' \ \{\overline{d}\} \quad O\, l^* \doteqdot u' \in \overline{d} \quad \Gamma \vdash u \leq_{\text{wf}} t}{\Gamma \vdash t@(u).l^* \text{ wf}} \qquad \text{(W-DLG)}$$

$$\frac{\Gamma \vdash t \leq_{\text{wf}} (: u \ \mathsf{inline}(\bar{s}) = u')}{\Gamma \vdash t\$ \text{ wf}} \qquad \text{(W-EXT)}$$

$$\frac{\Gamma \vdash G(\bar{t})_? \text{ wf} \quad \Gamma \vdash G(\bar{t}) \text{ wlk}}{\Gamma \vdash G(\bar{t})_\surd \text{ wf}} \qquad \text{(W-MOD}\surd\text{)}$$

Program well-formedness:

$$\frac{\overline{G} :: \emptyset \vdash \overline{M} \text{ wf} \quad \emptyset \vdash t \equiv_{\text{wf}} s^k \quad \text{There are no illegal cycles.}}{\mathsf{let}\ \overline{G} = \overline{M}\ \mathsf{in}\ t \text{ wf}} \qquad \text{(W-PROG)}$$

Module wf: $\boxed{G(\bar{s}) :: \Gamma \vdash M \text{ wf}}$

$$\frac{G(\bar{s}, x) :: \Gamma, x \leq t \vdash M \text{ wf} \quad x \notin \text{npvars}(*, M)}{G(\bar{s}) :: \Gamma \vdash \lambda^* x \leq t.\, M \text{ wf}} \qquad \text{(W-MFUN)}$$

$$\frac{\Gamma \vdash t \text{ wf} \quad \Gamma, x \leq G(\bar{s})_? \vdash \overline{d} \text{ wf} \quad \overline{d} \text{ has no duplicate labels.}}{G(\bar{s}) :: \Gamma \vdash \mu x \ \mathsf{extends}\ t\ \{\overline{d}\} \text{ wf}} \qquad \text{(W-MDEF)}$$

Declaration well-formedness: $\boxed{\Gamma \vdash d \text{ wf}}$

$$\frac{x \notin \text{npvars}(*, \mathsf{def}\ l^* \doteqdot t) \quad \Gamma, x \leq G(\bar{s})_\sigma \vdash t \text{ wf}}{\Gamma, x \leq G(\bar{s})_\sigma \vdash \mathsf{def}\ l^* \doteqdot t \text{ wf}} \qquad \text{(W-DECL)}$$

$$\frac{x \notin \text{pvars}(*, \mathsf{override}\ l^* \doteqdot t) \quad G(\bar{s}) \rightsquigarrow \mu y \ \mathsf{extends}\ u\ \{\overline{d}\} \quad \Gamma, x \leq G(\bar{s})_\sigma \vdash t \leq_{\text{wf}} u@(x).l^*}{\Gamma, y \leq G(\bar{s})_\sigma \vdash \mathsf{override}\ l^* \doteqdot t \text{ wf}} \qquad \text{(W-ODECL)}$$

Figure A.11: The DEEP calculus — well-formedness

Well-linked terms:

$$
\begin{array}{rll}
t^k, u^k, s^k ::= & \text{(well-linked terms)} \\
& x \\
& \mathsf{Top} \\
& \lambda^*_{\surd} x \le t^k.\ u^k \\
& \lambda^*_{?} x \le t^k.\ t \\
& G(\overline{t^k})_{\surd} \\
& : t^k\ \mathsf{inline}(\overline{s^k}) = u^k \\
& t^k (u^k)^*_{\surd} \\
& t^k @ (u^k).l^* \\
& t^k \$
\end{array}
$$

Well-linked modules: $\boxed{\Gamma \vdash\ G(\overline{t})\ \text{wlk}}$

$$
\frac{\begin{array}{c} G(\overline{t}) \rightsquigarrow \mu x\ \mathsf{extends}\ u\ \{\overline{d}\} \\ (\mathsf{def}\ l \doteqdot s) \in \overline{d} \quad \text{implies } \Gamma \vdash\ u.l\ \text{undefined} \\ (\mathsf{override}\ l \doteqdot u) \in \overline{d}\ \text{implies } \Gamma \vdash\ u.l\ \text{virtual} \\ \Gamma \vdash\ G(\overline{t})_{\surd} @ (G(\overline{t}))\ \text{valid} \end{array}}{\Gamma \vdash\ G(\overline{t})\ \text{wlk}}
$$

$$
\Gamma \vdash\ \mathsf{Top} @ (G(\overline{s}))\ \text{valid}
$$

$$
\frac{\begin{array}{c} \Gamma \vdash\ t \equiv H(\overline{u})_{\surd} \\ H(\overline{u}) \rightsquigarrow \mu x\ \mathsf{extends}\ t'\ \{\overline{O\ l \doteqdot u}\} \\ \forall u_i.\ \Gamma \vdash\ [x \mapsto G(\overline{s})_{\surd}] u_i \equiv s_i^k \\ \Gamma \vdash\ t' @ (G(\overline{s}))\ \text{valid} \end{array}}{\Gamma \vdash\ t @ (G(\overline{s}))\ \text{valid}}
$$

$$
\Gamma \vdash\ \mathsf{Top}.l\ \text{virtual}
$$

$$
\Gamma \vdash\ \mathsf{Top}.l\ \text{undefined}
$$

$$
\frac{\begin{array}{c} \Gamma \vdash\ t \equiv G(\overline{s})_{\surd} \\ G(\overline{s}) \rightsquigarrow \mu x\ \mathsf{extends}\ t'\ \{\overline{d}\} \\ O\ l^* = u\ \notin \overline{d} \quad \Gamma \vdash\ t'.l\ \text{virtual} \end{array}}{\Gamma \vdash\ t.l\ \text{virtual}}
$$

$$
\frac{\begin{array}{c} \Gamma \vdash\ t \equiv G(\overline{s})_{\surd} \\ G(\overline{s}) \rightsquigarrow \mu x\ \mathsf{extends}\ t'\ \{\overline{d}\} \\ l \notin \mathrm{dom}(\overline{d}) \quad \Gamma \vdash\ t'.l\ \text{undefined} \end{array}}{\Gamma \vdash\ t.l\ \text{undefined}}
$$

$$
\begin{array}{llll}
\mathrm{npvars}(\pm, t) = \emptyset & & \mathrm{npvars}(\pm, M) = \emptyset & \\
\mathrm{npvars}(+, t) = \mathrm{IV}(t) & & \mathrm{npvars}(+, M) = \mathrm{IV}(M) &
\end{array}
$$

$$
\begin{array}{lcll}
\mathrm{IV}(x) & = & \emptyset & \\
\mathrm{IV}(\mathsf{Top}) & = & \emptyset & \\
\mathrm{IV}(t(u)^{\pm}) & = & \mathrm{IV}(t) \cup \mathrm{fv}(u) & \\
\mathrm{IV}(t(u)^{+}) & = & \mathrm{IV}(t) \cup \mathrm{IV}(u) & \\
\mathrm{IV}(t @ (u).l^{\pm}) & = & \mathrm{IV}(t) \cup \mathrm{fv}(u) & \\
\mathrm{IV}(t @ (u).l^{+}) & = & \mathrm{IV}(t) \cup \mathrm{IV}(u) & \\
\mathrm{IV}(\lambda^*_{\sigma} x \le t.\ u) & = & \mathrm{fv}(t) \cup \mathrm{IV}(u) - \{x\} & \\
\mathrm{IV}(:\ t\ \mathsf{inline}(\overline{s}) = u) & = & \mathrm{fv}(t) & \\
\mathrm{IV}(G()) & = & \emptyset & \\
\mathrm{IV}(G(\overline{t}, u)) & = & \mathrm{IV}(G(\overline{t})) \cup \mathrm{fv}(u) & \text{if } G(\overline{t}) \rightsquigarrow \lambda x^{\pm} \le s.\ M \\
\mathrm{IV}(G(\overline{t}, u)) & = & \mathrm{IV}(G(\overline{t})) \cup \mathrm{IV}(u) & \text{if } G(\overline{t}) \rightsquigarrow \lambda x^{+} \le s.\ M
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{IV}(\lambda^* x \le t.\ M) & = & \mathrm{fv}(t) \cup \mathrm{IV}(M) - \{x\} \\
\mathrm{IV}(\mu x\ \mathsf{extends}\ t\ \{\overline{d}\}) & = & \mathrm{IV}(t) \cup \mathrm{IV}(\overline{d}) - \{x\} \\
\mathrm{IV}(O\ l^* : t) & = & \mathrm{IV}(t) \\
\mathrm{IV}(O\ l^* = t) & = & \mathrm{fv}(t)
\end{array}
$$

The set $\mathrm{IV}(t)$ is the set of free variables that are in invariant positions in t. (We write $\mathrm{IV}(t)$ in capital letters to distinguish it visually from $\mathrm{fv}(t)$, which is the set of all free variables in t.)

Figure A.12: The DEEP calculus — well-linkedness and invariant variables

Bibliography

M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996a.

Martín Abadi and Luca Cardelli. On subtyping and matching. *ACM Transactions on Programming Languages and Systems*, 18(4), 1996b.

Pierre America. Designing an object-oriented programming language with behavioural subtyping. *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, 1991.

D. Ancona and E. Zucca. A calculus of module systems. *Journal of Functional Programming*, 12(2):91–132, 2002.

Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam - a smooth extension of java with mixins. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2000.

Zena Ariola and Stefan Blom. Cyclic lambda calculi. *Theoretical Aspects of Computer Software*, 1997.

Zena Ariola and Jan Willem Klop. Lambda calculus with explicit recursion. *Information and Computation*, 139, 1997.

David Aspinall. Subtyping with singleton types. *Eighth International Workshop on Computer Science Logic*, 1994.

David Aspinall. Subtyping with power types. *Proceedings of Computer Science Logic*, 2000.

David Aspinall and Adriana Compagnoni. Subtyping dependent types. *Proceedings of 11th Annual Symposium on Logic in Computer Science*, 1996.

Lennart Augustsson. Cayenne - a language with dependent types. *Proceedings of the International Conference on Functional Programming (ICFP)*, 1998.

Lennart Augustsson and Magnus Carlsson. An exercise in dependent types: A well-typed interpreter. *Proceedings of the Workshop on Dependent Types in Programming*, 1999.

Henk Barendregt. Lambda calculi with types. *Handbook of Logic in Computer Science, volume II*, 1992.

Kim Barrett et al. A monotonic superclass linearization for dylan. *Proceedings of OOPSLA*, 1996.

D. Batory, J. Liu, and J. Sarvela. Refinements and multi-dimensional separation of concerns. *ACM SIGSOFT*, 2003.

D. Batory, J. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 2004.

Don Batory and Sean OMalley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1992.

A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Classboxes: Controlling visibility of class extensions. *Computer Languages, Systems and Structures*, 31(3):107–126, 2005.

Jan Bergstra and Jan Klop. Conditional rewrite rules: Confluence and termination. *Journal of Computer and System Sciences*, 1986.

Marc Bezem, Jan Willem Klop, and editors Roel de Vrijer. *Term Rewriting Systems*. Number 55. Cambridge Tracts in Theoretical Computer Science, 2003.

G. Blaschek. Type-safe OOP with prototypes: the concepts of Omega. *Structured Programming*, 12(12):1–9, 1991.

Daniel Bobrow, Linda DeMichiel, Richard Gabriel, Sonya Keene, Gregor Kiczales, and David Moon. Common lisp object system specification. *ACM SIGPLAN Notices*, 23, 1988.

G. Bracha and W. Cook. Mixin-based inheritance. *Proceedings of OOPSLA*, 1990.

Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance.* PhD thesis, University of Utah, 1992.

Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. *Proceedings of the IEEE Conference on Computer Languages*, 1992.

Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. *Types for Proofs and Programs*, 2003.

K.B. Bruce. Some challenging typing issues in object-oriented languages. *Electronic Notes in Theoretical Computer Science 82 No. 8*, 2003.

K.B. Bruce, A. Fiech, and L. Petersen. Subtyping is not a good "match" for object-oriented languages. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1997.

Kim Bruce, Luca Cardelli, Giuseppe Castagna, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3), 1995.

Kim Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1998.

Martin Buchi and Wolfgang Weck. Generic wrappers. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2000.

Peter Canning, William Cook, Walter Hill, and Walter Olthoff. F-bounded polymorphism for object-oriented programming. *Proceedings of Functional Programming Languages and Computer Architecture*, pages 273–280, 1989.

Luca Cardelli. Structural subtyping and the notion of power type. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 1988.

Luca Cardelli. Program fragments, linking, and modularization. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 1997.

Jacques Carette, Oleg Kiselyov, and Chung chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languges. *The 5th Asian Symposium on Programming Languages and Systems*, 2007.

Robert Cartwright and Mike Fagan. Soft typing. *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 1991.

Guiseppe Castagna and Gang Chen. Dependent types with subtyping and late-bound overloading. *Information and Computation*, 168, 2001.

Manuel Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2005.

Gang Chen. Subtyping calculus of constructions (extended abstract). *Proceedings of the International Symposium on Mathematical Foundations of Computer Science*, 1997.

Gang Chen. Dependent type system with subtyping: Type level transitivity elimination. *Journal of Computer Science and Technology*, 14(1), 1999.

James Cheney and Ralf Hinze. First-class phantom types. *Cornell University Technical Report TR2003-1901*, 2003.

Niels Christensen, Robert Glück, and Søren Laursen. Binding-time analysis in partial evaluation: One size does not fit all. *Proceedings of the Third International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, 2000.

Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 1940.

Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. Tribe: A simple virtual class calculus. *Proceedings of the Conference on Aspect-Oriented Software Development (AOSD)*, 2007.

Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: modular open classes and symmetric multiple dispatch for java. *Proceedings of OOPSLA*, 2000.

Dario Colazzo and Giorgio Ghelli. Subtyping recursive types in kernel fun. *Logic in Computer Science*, 1999.

Adriana Compagnoni. *Higher-Order Subtyping with Intersection Types*. PhD thesis, University of Nijmegen, 1995.

Adriana Compagnoni and Healfdene Goguen. Typed operational semantics for higher order subtyping. *Information and Computation*, 184(2), 2003.

Adriana Compagnoni and Benjamin Pierce. Higher-order intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 6(5), 1996.

Charles Consel. A tour of schism: a partial evaluation system for higher-order applicative languages. *Proceedings of the ACM Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 1993.

William Cook. Object-oriented programming versus abstract data types. *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, 1990.

William Cook, Walter Hill, and Peter Canning. Inheritance is not subtyping. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 1990.

Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 1(2/3), 1988.

Patrick Cousot. Abstract interpretation based formal methods and future challenges. *Informatics, 10 Years Back, 10 Years Ahead*, 2001.

Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 1977.

Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 1999.

Krzysztof Czarnecki and Ulrich Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

Olivier Danvy. Type-directed partial evaluation. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 1996.

Olivier Danvy. A simple solution to type specialization. *Proceedings of the International Colloquium on Automata, Languages and Programming*, 1998.

Nicolaas Govert de Bruijn. The mathematical language automath, its usage and some of its extensions. *M. Laudet, D. Lacombe, and M. Schuetzenberger (editors), Symposium on Automatic Demonstration, Lecture Notes in Mathematics 125*, 1970.

Philippe de Groote. Definining λ-typed λ-calculi by axiomatizing the typing relation. *The Symposium on Theoretical Aspects of Computer Science*, 1993.

Phillipe de Groote (editor). *The Curry-Howard Isomorphism*. Cahiers du Centre de Logique (Universit catholique de Louvain), Academia-Bruylant, 1995.

Derek Dreyer. *Understanding and Evolving the ML Module System*. PhD thesis, Carnegie Mellon University, 2005.

Derek Dreyer. A type system for recursive modules. *Proceedings of the International Conference on Functional Programming (ICFP)*, 2007.

Derek Dreyer, Robert Harper, and Karl Crary. Toward a practical type theory for recursive modules. *Technical Report CMU-CS-01-112*, 2001.

Derek Dreyer, Robert Harper, and Manuel Chakravarty. Modular type classes. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2007.

Dominic Duggan and Constantinos Sourelis. Mixin modules. *Proceedings of the International Conference on Functional Programming (ICFP)*, 1996.

Margaret Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

Erik Ernst. Propagating class and method combination. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1999a.

Erik Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance.* PhD thesis, University of Aarhus, Denmark, 1999b.

Erik Ernst. Family polymorphism. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2001.

Erik Ernst. Higher order hierarchies. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2003.

Erik Ernst, Klaus Ostermann, and William Cook. A virtual class calculus. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2006.

Kathleen Fisher and John Mitchell. A delegation-based object calculus with subtyping. *Fundamentals of Computation Theory*, 1995.

M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 1998.

Matthew Flatt and Matthias Felleisen. Units: Cool modules for hot languages. *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 1998.

Yoshihiko Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4), 1999.

Jean Gallier. On girards "candidats de reductibilite". *Logic and Computer Science*, (31), 1990.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

Benedict Gaster and Mark Jones. A polymorphic type system for extensible records and variants. *Technical report NOTTCS-TR-96-3, University of Nottingham*, 1996.

Giorgio Ghelli. The s-all-loc variant of fsub. *TYPES mailing list*, February 24, 1993. URL `http://www.seas.upenn.edu/~sweirich/types/archive/1993/msg00040.html`.

Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris, 1972.

Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science, 1989.

Healfdene Goguen. Typed operational semantics. *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, 1995.

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification (3rd Edition)*. Prentice Hall, 2005.

Jr. Guy L. Steele. *Common LISP: the language (2nd ed.)*. Digital Press, Newton, MA, USA, 1990.

Robert Harper and Christopher Stone. A type-theoretic interpretation of standard ml. *Proof, language, and interaction: essays in honour of Robin Milner*, 2000.

Robert Harper, John Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 1989.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40, 1993.

James Hook and Douglas Howe. Impredicative strong existential equivalent to type:type. *Cornell University Technical report TR86-760*, 1986.

Douglas Howe. The computational behavior of girard's paradox. *Proceedings of the Symposium on Logic in Computer Science*, 1987.

Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Notices*, 27(5), 1992.

Paul Hudak, John Peterson, and Joseph Fasel. A gentle introduction to haskell 98. `http://www.haskell.org/tutorial/`, 1999.

John Hughes. Type specialization for the λ-calculus; or, a new paradigm for partial evaluation based on type inference. *Proceedings of The International Seminar on Partial Evaluation*, 1996.

DeLesley Hutchins. The power of symmetry: Unifying inheritance and generative programming. *OOPSLA Companion, DDD Track*, 2003.

DeLesley Hutchins. Eliminating distinctions of class: Using prototypes to model virtual classes. *Proceedings of OOPSLA*, 2006.

Atsushi Igarashi and Benjamin Pierce. Foundations for virtual types. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1999.

Atsushi Igarashi and Benjamin Pierce. On inner classes. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2000.

Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight java, a minimal core calculus for java and gj. *Proceedings of OOPSLA*, 1999.

Atsushi Igarashi, Chieri Saito, and Mirko Viroli. Lightweight family polymorphism. *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*, 2005.

Benedetto Intrigila, Ivano Salvo, and Stefano Sorgi. A characterization of weakly church-rosser abstract reduction systems that are not church-rosser. *Information and Computation*, 171(2), 2001.

Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, 1985.

P. Jolly, S. Drossopoulou, C. Anderson, and K. Ostermann. Simple dependent types: Concord. *Proceedings of the Workshop on Formal Techniques for Java-like Programs (FTfJP)*, 2005.

Mark P. Jones. A theory of qualified types. *Proceedings of the European Symposium on Programming*, 1992.

N.D. Jones. Mix ten years later. *Proceedings of the ACM Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 1995.

Neil Jones, Carsten Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1993.

Neil D. Jones and Arne J. Glenstrup. Program generation, termination, and binding-time analysis. *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, 2002.

Fairouz Kamareddine. Typed λ-calculi with one binder. *Journal of Functional Programming*, 15(5), 2005.

Fairouz Kamareddine and Rob Nederpelt. Canonical typing and π-conversion in the barendregt cube. *Journal of Functional Programming*, 6(2), 1996.

Fairouz Kamareddine, Roel Bloo, and Rob Nederpelt. On π-conversion in the λ-cube and the combination with abbreviations. *Annals of Pure and Applied Logic*, 1997.

G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2001.

Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The art of metaobject protocol*. MIT Press, Cambridge, MA, USA, 1991. ISBN 0-262-61074-4.

Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1997.

Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, 2004.

Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121 (1-2), 1993.

Jan Willem Klop, Vincent van Oostrom, and Roel de Vrijer. A geometric proof of confluence by decreasing diagrams. *Journal of Logic and Computation*, 10(3), 2000.

Günter Kniesel. *Dynamic Object-based Inheritance with Subtyping*. PhD thesis, Institut für Informatik Rheinische Friedrich-Wilhelms-Universität Bonn, 2000.

Shriram Krishnamurthi, Matthias Felleisen, and Daniel Friedman. Synthesizing object-oriented and functional design to promote re-use. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1998.

Chris Lattner. Llvm: An infrastructure for multi-stage optimization. Master's thesis, University of Illinois at Urbana-Champaign, 2002.

Xavier Leroy. Manifest types, modules, and separate compilation. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 1994.

John Levine, Tony Mason, and Doug Brown. *lex & yacc (2nd ed.)*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1992. ISBN 1-56592-000-7.

Jeffrey Lewis, Mark Shields, Erik Meijer, and John Launchbury. Implicit parameters: Dynamic scoping with static types. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2000.

Henry Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. *Proceedings of OOPSLA*, 1986.

Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, Carnegie Mellon University, 1996.

Jia Liu, Don Batory, and Srinivas Nedunuri. Modeling interactions in feature-oriented software designs. *Proceedings of the International Conference on Feature Interactions*, 2005.

R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating support for features in advanced modularization technologies. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2005.

Roberto Lopez-Herrejon, Don Batory, and Christian Lengauer. A disciplined approach to aspect composition. *Proceedings of the ACM Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 2006.

Yong Luo and Zhaohui Luo. Coherence and transitivity in coercive subtyping. *Proceedings of the 8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, 2001.

Zhaohui Luo. *Computation and reasoning: a type theory for computer science*. Oxford University Press, Inc., New York, NY, USA, 1994. ISBN 0-19-853835-9.

Zhaohui Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1), 1999.

Zhaohui Luo and Randy Pollack. Lego proof development system: User's manual. *LFCS Technical Report ECS-LFCS-92-211*, 1992.

David MacQueen. Modules for standard ml. *Proceedings of the ACM Symposium on LISP and functional programming*, 1984.

David MacQueen. Using dependent types to express modular structure. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 1986.

O.L. Madsen and B. Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. *Proceedings of OOPSLA*, 1989.

Henning Makholm. On Jones-optimal specialization for strongly-typed languages. *Proceedings of the Workshop on Semantics, Applications, and Implementations of Program Generation*, 2000.

John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 1998.

Simon Marlow and Simon Peyton Jones. The new ghc/hugs runtime system. *Unpublished. Available from* `http://research.microsoft.com/users/simonpj/Papers/papers.html`, 1998.

Per Martin-Löf. Intuitionistic type theory. *Bibliopolis*, 1984.

Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1), 2004.

John Mitchell, Sigurd Meldal, and Neel Madhav. An extension of standard ml modules with subtyping and inheritance. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 1991.

Keiko Nakata and Jacques Garrigue. Recursive modules for programming. *Proceedings of the International Conference on Functional Programming (ICFP)*, 2006.

Flemming Nielson and Hanne Riis Nielson. *Two-level functional languages*. Cambridge University Press, New York, NY, USA, 1992. ISBN 0-521-40384-7.

Nathaniel Nystrom, Stephen Chong, and Andrew Myers. Scalable extensibility via nested inheritance. *Proceedings of OOPSLA*, 2005.

Nathaniel Nystrom, Xin Qi, and Andrew Myers. J&: Nested intersection for scalable software composition. *Proceedings of OOPSLA*, 2006.

Martin Odersky and et. al. *The Scala Language Specification*. `http://scala.epfl.ch`, 2004.

Martin Odersky and Matthias Zenger. Scalable component abstractions. *Proceedings of OOPSLA*, 2005.

Martin Odersky, Vincent Cremet, Christine Rockl, and Matthias Zenger. A nominal theory of objects with dependent types. *Proceedings of ECOOP*, 2003.

Klaus Ostermann. Dynamically composable collaborations with delegation layers. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2002.

Emir Pašalić, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. *Proceedings of the International Conference on Functional Programming (ICFP)*, 2002.

W. Harrison Peri Tarr, H. Ossher. N degrees of separation: Multi-dimensional separation of concerns. *Proceedings of the International Conference on Software Engineering (ICSE)*, 1999.

Simon Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. *Journal of Functional Programming*, 1992.

Simon Peyton Jones and Erik Meijer. Henk: a typed intermediate language. *Workshop on Types in Compilation*, 1997.

Simon Peyton Jones, Mark P. Jones, and Erik Meijer. Type classes: exploring the design space. *Proceedings of the Haskell Workshop, Amsterdam, The Netherlands*, 1997.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for gadts. *Proceedings of the International Conference on Functional Programming (ICFP)*, 2006.

Frank Pfenning and Carsten Schürmann. System description: Twelf, a meta-logical framework for deductive systems. *Proceedings of the International Conference on Automated Deduction*, 1999.

Benjamin Pierce. Bounded quantification is undecidable. *Information and Computation*, 1994.

Benjamin Pierce. *Types and Programming Languages*. MIT Press, 2002.

Gordon Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, pages 125–159, 1975.

Elke Pulvermuellern, Andreas Speck, James Coplien, Maja DHondt, and Wolfgang De Meuter. Feature interaction in composed systems. *Object Oriented Technology: ECOOP 2001 Workshop Reader*, 2002.

Derek Rayside and Gerard T. Campbell. An aristotelian understanding of object-oriented programming. *Proceedings of OOPSLA*, 2000.

John Reynolds. User-defined types and procedural data as complementary approaches to data abstraction. *In S. A. Schuman (editor), New Directions in Algorithmic Languages, IFIP Working Group 2.1 on Algol*, 1975.

Claudio Russo. First-class structures for standard ML. *Nordic Journal of Computing*, 2000.

N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2002.

Tim Sheard. Languages of the future. *Proceedings of the OOPSLA Onward Track*, 1994.

Tim Sheard. Putting curry howard to work. *Proceedings of the ACM Workshop on Haskell*, 2005.

Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. *The Haskell Workshop*, 2002.

Y. Smaragdakis and D. Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2002.

Lee Smolin. *Three Roads to Quantum Gravity*. Perseus Books Group, 2002. ISBN 978-0465078363.

Martin Steffen. *Polarized Higher-Order Subtyping*. PhD thesis, Universität Erlangen-Nürnberg, 1997.

Martin Steffen and Benjamin Pierce. Higher-order subtyping. *University of Edinburgh Technical Report ECS-LFCS-94-280*, 1994.

Christopher Stone. *Singleton Kinds and Singleton Types*. PhD thesis, Carnegie Mellon University, 2000.

Martin Sulzmann, Manuel Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System f with type equality coercions. *Proceedings of the International Workshop on Types in Languages Design and Implementation*, 2007.

Eijiro Sumii and Naoki Kobayashi. Online-and-offline partial evaluation: a mixed approach. *Proceedings of the ACM Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 1999.

Wouter Swierstra. Data types á la carte. *Journal of Functional Programming*, 2008.

Clemens Szyperski. Import is not inheritance, why we need both: Modules and classes. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1992.

Walid Taha. A gentle introduction to multi-stage programming. *Domain-Specific Program Generation*, 2003.

Walid Taha and Henning Makholm. Tag elimination – or – type specialization is a type-indexed efect. *Subtyping and Dependent Types in Programming, APPSEM Workshop, INRIA technical report*, 2000.

Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1–2), 2000.

Walid Taha, Henning Makholm, and John Hughes. Tag elimination and jones optimality. *Proceedings of the Second Symposium on Programs as Data Objects*, 2001.

William Tait. A realizability interpretation of the theory of species. *Logic Colloquium*, 1975.

Antero Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3), 1996.

Masako Takahashi. Parallel reductions in λ-calculus. *Informationa and Computation*, 118(1), 1995.

The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 8.1*. INRIA, 2006.

R. D. Tennent. The denotational semantics of programming languages. *Communications of the ACM*, 19(8), 1976.

K. K. Thorup and M. Torgersen. Unifying genericity — combining the benefits of virtual types and parameterized classes. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1999.

Mads Tofte. Essentials of standard ml modules. *Advanced Functional Programming, Second International School-Tutorial Text*, 1129, 1996.

Mads Torgersen. Virtual types are statically safe. *5th Workshop on Foundations of Object-Oriented Languages*, 1998.

Mads Torgersen. The expression problem revisited. four new solutions using generics. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2004.

Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the java programming language. *In Journal of Object Technology*, 3(11), 2004.

David Ungar and Randall Smith. Self, the power of simplicity. *Proceedings of OOPSLA*, 1987.

Vincent van Oostrom. Confluence by decreasing diagrams. *Theoretical Computer Science*, 126(1), 1994.

Todd Veldhuizen. C++ templates as partial evaluation. *Proceedings of the ACM Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 1999.

Todd L. Veldhuizen. Arrays in blitz++. *Proceedings of Scientific Computing in Object-Oriented Parallel Environments (ISCOPE)*, 1998.

Philip Wadler. Recursive types for free! *Unpublished draft, available as* `http://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt`, 1990.

Philip Wadler. The essence of functional programming. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 1992.

Philip Wadler. The expression problem. *posted to Java Genericity internet mailing list*, 1998.

Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 1989.

Steven Weinberg. *Dreams of a Final Theory: The Scientist's Search for the Ultimate Laws of Nature*. Vintage, 1994. ISBN 978-0679744085.

J. B. Wells and René Vestergaard. Equational reasoning for linking with first-class primitive modules. *Proceedings of the European Symposium on Programming Languages and Systems*, 2000.

Joe Wells, Detlef Plump, and Fairouz Kamareddine. Diagrams for meaning preservation. *Rewriting Techniques and Applications*, 2003.

Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 2004.

HongWei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998.

Hongwei Xi. Facilitating program verification with dependent types. *Proceedings of the International Conference on Software Engineering and Formal Methods*, 2003.

Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2003.

Matthias Zenger and Martin Odersky. Extensible algebraic datatypes with defaults. *Proceedings of the International Conference on Functional Programming (ICFP)*, 2001.

Matthias Zenger and Martin Odersky. Independently extensible solutions to the expression problem. *Workshop on Foundations of Object-Oriented Languages*, 2005.

Jan Zwanenburg. Pure type systems with subtyping. *International Conference on Typed Lambda Calculi and Applications*, 1999.